



Universidade de Aveiro
2009

Departamento de Electrónica, Telecomunicações e
Informática (DETI)

Sérgio Torres Soldado

**SIMULAÇÃO E TESTE DE ELEMENTOS DE
UMA REDE DE TRÁFEGO URBANA EM
FPGA**



Universidade de Aveiro
2009

Departamento de Electrónica, Telecomunicações e
Informática (DETI)

Sérgio Torres Soldado

**FPGA URBAN TRAFFIC CONTROL
SIMULATION AND EVALUATION
PLATFORM**



Universidade de Aveiro
2009

Departamento de Electrónica, Telecomunicações e
Informática (DETI)

**FPGA URBAN TRAFFIC CONTROL
SIMULATION AND EVALUATION PLATFORM**

by

Sérgio Torres Soldado

A thesis submitted in partial fulfillment of the requirements
for the degree of

Electronic and Telecommunications Engineering

Universidade de Aveiro

June 15, 2009

O Júri / The Jury

Presidente / President

**Prof. Dr. António de Brito
Ferrari**

Professor Catedrático da Universidade de Aveiro

Arguentes / Examiners

Prof. Dr. Hélio Mendonça

Professor da Faculdade de Engenharia da Universidade do Porto

Prof. Dr. Valeri Skliarov

Professor Catedrático da Universidade de Aveiro

ABSTRACT

FPGA Urban Traffic Control Simulation and Evaluation Platform

by Sérgio Torres Soldado

The study and development towards Urban Traffic Management and Control (UTMC) Systems have not solely or recently gained extreme importance only due to obvious issues such as traffic safety improvement, traffic congestion control and avoidance but also due to other underlying factors such as urban transportation efficiency, urban traffic originated air pollution and future concepts as are autonomous vehicle systems, which are presently taking shape. Generally speaking urban traffic simulations occur in a software environment, which comes to hinder the progress taken towards the actual implementation of UTMC systems. The reason to why such happens is based on the fact that urban traffic controllers are usually implemented and executed on hardware platforms, therefore software based models don't support an actual implementation directly. In this study we explore a novel approach to urban traffic simulation, aimed to eliminate the timeframe and work-distance between the UTMC system's design and an eventual implementation, where a Field Programmable Gate Array (FPGA) is used to execute a simulation model of an urban traffic network. Since the resource to FPGAs implies a hardware based execution, the resulting implementation of each traffic management and control element can be considered not only as having a close matched behavior to a real world implementation but also as an actual prototype. From the simulation viewpoint the use of FPGA's holds the prospect of being able to hold execution speeds many times faster than software based simulations as FPGA designs are able to execute a large number of parallel processes. This study shows that an Urban Traffic Control Simulation and Test Platform is possible by implementing a relatively simple urban network model in a low end FPGA. This result implies that with further time and resource investments a rather complex system can be developed which can handle large scale and complex UTMC systems with the promise of shortening the work distance between the concept and a real world running implementation.

TABLE OF CONTENTS

List of Figures.....	ii
List of Tables.....	iii
List of Code Descriptions.....	iii
1 Preface.....	1
1.1 Introduction.....	1
1.2 Background.....	2
1.2.1 Urban Traffic Control Systems.....	2
1.2.2 Micro-Simulation.....	4
1.3 Motivation.....	4
1.4 Related Work.....	7
1.5 Thesis Outline.....	8
2 Development Tools.....	10
Overview.....	10
2.1 Hardware.....	10
2.1 Software.....	13
3 Detailed Implementation.....	15
Overview.....	15
3.1 Architecture: Top Level.....	15
3.2 Peripheral Controllers.....	18
3.2.1 VGA Controller.....	18
3.2.2 PS/2 Mouse Module.....	19
3.2.3 FLASH Memory Controller.....	21
3.3 UTS Network:.....	28
3.3.1 Architecture.....	28
3.3.2 Network Infrastructure Data Set.....	32
3.4 Traffic Lights.....	38
3.4.1 Traffic Light Model Elements and Data Set.....	38
3.4.2 Basic Traffic Light Model.....	42
3.4.3 Intelligent Traffic Light Model.....	51
3.5 Vehicles.....	53
3.5.1 Vehicle Model Elements and Data Set.....	53
3.5.2 “Dummy” Vehicle Model.....	56
3.5.3 User Controlled Vehicle Model.....	64
i) Vehicle Controller.....	66
ii) Dijkstra’s Algorithm.....	70
iii) User Input Controller.....	76
3.6 User Interface.....	80
3.6.1 Text Generation.....	90

3.7	VGA Output Components	96
3.7.1	Traffic Lights	96
3.7.2	Vehicles	98
4	Design Flow	101
5	Implementation Results & Analysis	104
5.1	Encoding.....	104
5.2	Dijkstra's Algorithm	111
5.3	Simulation Results	113
5.4	Debugging Results	115
6	Conclusions & Future Work.....	116
A	Appendix	119
	Shortest Path Finding in Matlab	119
B	Bibliography.....	128

LIST OF FIGURES

<i>Number</i>	<i>Page</i>
Figure 1.3-1 Evolution of FPGA architectures, figure extracted from [7], pertaining to Xilinx	5
Figure 2.1-1 Development system hardware setup	11
Figure 2.1-2 - Digilent Nexys2 Board features overview, figure extracted from the Nexys2 Reference Manual available at the Digilent website.....	12
Figure 3.1-1 Top Level System Architecture	16
Figure 3.1-2 System clock signals generation with DCMs	17
Figure 3.2.1-1 VGA Synchronization Circuit	18
Figure 3.2.2-1 Mouse Module Architecture	19
Figure 3.2.3-1 Flash memory controller.....	22
Figure 3.2.3-2 Image to FLASH bitmap conversion function.....	23
Figure 3.2.3-3 FLASH memory addressing with offset mechanism.....	25
Figure 3.2.3-4 Tile map used to draw the network road map	26
Figure 3.2.3-5 Tiled: tile-map editor being executed	27
Figure 3.2.3-6 Graphical representation of an UTS network road map where the intersections have been numbered for identification.....	28
Figure 3.3.1-1 UTS Network Module Architecture.....	31
Figure 3.3.2-1 UTS network infrastructure data set	32
Figure 3.3.2-2 Intersection describing example.....	34
Figure 3.3.2-3 Intersection position coordinates memory component.....	37
Figure 3.4.1-1 Timer values for each traffic light memory component	40
Figure 3.4.1-2 Timeout values for each traffic light memory component.....	41
Figure 3.4.1-3 Timer RAM initialization of first intersection	42
Figure 3.4.1-4 Timeout RAM initialization of first intersection.....	42
Figure 3.4.2-1 Cross-linked traffic light scheme.....	43
Figure 3.4.2-2 Independent cycle for each route traffic light scheme	44
Figure 3.4.2-3 Traffic light control state machine diagram	50
Figure 3.4.3-1 - Intelligent traffic light state machine diagram.....	52
Figure 3.5.1-1 Basic vehicle model data set elements.....	54
Figure 3.5.1-2 Clock divider/tick generation circuit.....	55
Figure 3.5.2-1 Dummy vehicle state chart.....	63
Figure 3.5.2-2 Illustration of intersection safety distance, in this case 16 pixels.....	64
Figure 3.5.3-1 User controlled vehicle activity diagram.....	66
Figure 3.5.3-2 User controlled vehicle movement FSM's state chart	69
Figure 3.5.3-3 Dijkstra's Algorithm Pseudo-Code, extracted from http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm	71
Figure 3.5.3-4 Procedure to extract shortest path sequence, extracted from http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm	71
Figure 3.5.3-5 – Dijkstra previous node memory component.....	72
Figure 3.5.3-6 – Dijkstra distance to target value memory component	72
Figure 3.5.3-7 Dijkstra's Algorithm state chart.....	73
Figure 3.5.3-8 User Input Controller state chart	79
Figure 3.6-1 Default screen with user interface menu to the right.....	82
Figure 3.6-2 User “INTERSECTIONS” interface	83

Figure 3.6-3 User “DEBUGGER” interface.....	84
Figure 3.6-4 Traffic Light Controller with user input	85
Figure 3.6-5 Screen divided in 16-by-16 pixel tiles in order to localize each button	89
Figure 3.6.1-1 Text Generation Module	91
Figure 3.6.1-2 ASCII to custom code translation function source code.....	93
Figure 3.6.1-3 Sample output for ASCII to custom code translation function	94
Figure 3.6.1-4 Text generation circuit memory addressing.....	95
Figure 3.6.1-5 Tile map area overlapping video screen	95
Figure 3.7.1-1 – Intersection traffic light position	96
Figure 3.7.1-2 Intersection position coordinates CAM	97
Figure 3.7.2-1 Dummy Vehicle CAM component	99
Figure 5.1.1-1 Synthesis and Implementation completion time in function of the number of vehicles in the UTS simulation.....	108
Figure 5.1.1-2 Maximum design operating frequency in function of the number of vehicles in the UTS simulation	109
Figure 5.1.1-3 – Resource usage in function of the number of vehicles in the UTS Simulation.....	110

LIST OF TABLES

Table 2.1-1 – Xilinx FPGA Family comparison including Spartan 3E-500 specifics	12
Table 3.4.2-1 Traffic light controller parameters for the first traffic light scheme.....	45
Table 3.4.2-2 Traffic light controller parameters for the second traffic light arrangement.....	46
Table 3.4.2-3 – Traffic light FSM state description.....	49
Table 3.6.1-1 Custom font character map	92
Table 5.1.1-1 Synthesis and Implementation Options	106

LIST OF CODE DESCRIPTIONS

VHDL Description 3.2.2-1 – Cursor ROM contents	20
VHDL Description 3.3.2-1 Intersection data structure	35
VHDL Description 3.3.2-2 UTS Network Infrastructure description example of first five intersections.....	36
VHDL Description 3.3.2-3 Intersection position coordinates RAM initialization file's first five entries.....	37
VHDL Description 3.4.1-1 Intersection traffic light controller VHDL data structure	40
VHDL Description 3.4.1-2 Intersection traffic control description.....	41
VHDL Description 3.5.1-1 Vehicle data set	56
VHDL Description 3.5.2-1 ROM used to identify right-hand side route	64
VHDL Description 3.5.3-1 Code description for the user controlled vehicle	67
VHDL Description 3.5.3-2 Dijkstra's Algorithm data signals	72
VHDL Description 3.6-1 User parameters loading FSM.....	87
VHDL Description 3.6-2 User interface input FSM.....	90
VHDL Description 3.7.1-1 Intersection position coordinates CAM.....	98
VHDL Description 3.7.2-1 User controlled vehicle VGA output signal	99
VHDL Description 3.7.2-2 RGB output multiplexer	100

ACKNOWLEDGMENTS

First off I would like to thank my professor and advisor Dr. Valeri Skliarov, who helped me build the necessary knowledge to complete this task. I also thank him for contributing greatly to my fulfillment during my attendance at the Universidade de Aveiro, one which will no doubtfully remain memorable during my life.

I thank my parents for the never ending support, love and friendship, which define my being and which I am very grateful to have. Mom and Dad, you are the people I most admire and love.

I would like to offer my deepest appreciation to my sister and to my friend Ana Guimarães for the ongoing support and endurance.

I would also like to cite “hats off” to my university colleagues in particular Abílio for being such a great friend.

ACRONYMS & DEFINITIONS

ASIC	Application Specific Integrated Circuit
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GUI	Graphical User Interface
HDL	Hardware Description Language
IP	Intellectual Property
Picoblaze	A series of three free soft processor cores from Xilinx for use in their FPGA and CPLD products
Pipeline	Set of data processing elements connected in series so that the output of one element is the input of the next one
RAM	Random Access Memory
UTC	Urban Traffic Control
UTMC	Urban Traffic Management and Control
UTS	Urban Traffic System
VGA	Video Graphics Array
VHDL	Very High Speed Integrated Circuit Hardware Description Language
Xilinx	World's largest supplier of programmable logic devices and the inventor of the FPGA

1 P R E F A C E

1.1 INTRODUCTION

In regard to the actual global environment, energy and economic crisis, it has been stated that traffic flows account for as much as one-third of the global energy consumption [1]. Recent studies reveal that unconventional changes in managing traffic flow can significantly lower harmful CO₂ emissions. An interesting perspective is that in which the reason behind hybrid vehicles is somehow contested as there is at least one study[2] which reaches the conclusion that intelligent vehicles (telematics-enabled vehicles) are capable of a greater economy than their hybrid counter-parts, without having the increased initial vehicle cost. Considering the outlook these matters might have in a near future at a global level, it can be stated that research and development taken towards innovative traffic management systems are subjects of major importance.

Because of the inherited non linear behavior of Urban Traffic Management and Control (UTMC) systems (e.g. a traffic lights can have various policies in function of time, traffic load and others.), practical traffic analysis often occurs in the form of a model which often gives place to graphical simulations. Although computer aided simulations of Urban Traffic Systems (UTS) can be dated back to 1955[6], it has only been in the last decade or so that major advances have allowed these simulations to form the basis of policy making. Such advances are due to major developments and promotions in traffic theory, computer hardware technology, programming tools and paradigms and most importantly due to social and economical factors.

As progress continues and applications grow in complexity new needs are in order. Regarding these needs, presently there has been an inclination towards designing and developing Urban Traffic Management and Control systems in a decentralized manner, which contrasts with the until-recently followed centralized approach. A detailed comparison of both centralized and decentralized architectures is given in the following section however the basic reason behind the inclination towards a decentralized architecture is due to the limitations regarding the conception and implementation of centralized systems impose when handling complex urban network

configurations. The nature of these limitations frequently lies in the fact that it is generally difficult to design and implement a single, monolithic system that extends itself over a wide geographical area monitoring and controlling a large number (possibly thousands) of traffic control and shaping elements.

Although there are presently a comprehensive number of increasingly competent UTS simulation tools, it is important to state that none could be found to comprise urban traffic management controllers directly, i.e. there are no simulation platforms dedicated to supporting UTMC systems which execute on actual real controllers and systems. In the context of developing an urban traffic simulation platform based on a decentralized architecture, and which has the capability of integrating of coupling real (i.e. not software based) traffic controllers, this paper suggests a hardware based implementation of an UTS simulator in order to:

- Develop and Evaluate UTMC systems through simulation in pseudo-real environments.
- Design and Evaluate complex traffic or vehicle controllers.

Based on this approach, the present work suggests the use of a Field Programmable Gate Array (**FPGA**) development board as the hardware platform on which the design will operate.

1.2 BACKGROUND

1.2.1 URBAN TRAFFIC CONTROL SYSTEMS

Urban traffic control (**UTC**) systems are a form of traffic management which co-ordinates traffic signal control over a wide area, normally where traffic flow is high. The goals are to condition and optimize this traffic flow as benefits are mainly achieved by the progression of traffic in an organized manner. As stated before, UTC systems were until recently generally based on a centralized conception, which relies on a central computer that communicates individually with each traffic controller element. This centralized approach has recently deprecated, since it has become

increasingly complex or even impossible to conceptualize such a model due to large systems, which have many traffic controlling elements such as traffic lights, vehicle passage and speed detectors. Since there are many different approaches to a centralized traffic control system incompatibility issues are also a concern, i.e. different technologies along with different communication protocols and standards can impose themselves as obstacles in developing and extending existent, or designing new UTMC systems. On the other hand the concept of a decentralized system breaks down hierarchy hereby suggesting that each traffic control element has an independent, although limited scope on the rest of the traffic control network. Data input and communication with the rest of the network's control elements are also considered to be minimal as to reduce hardware requirements so as to make an actual implementation feasible. The decentralized approach allows a generally less complex overall system design and implementation as otherwise factors such as the inheriting real time constraints in traffic control systems (communications and synchronization between varying traffic signals) would imply complicated negotiation mechanisms in the centralized approach because of shared communication channels. Consequently, from a decentralized viewpoint, each element such as each traffic light in an intersection must have a predefined behavior in function of how it perceives local traffic and the other traffic lights (in the same intersection). This behavior can also be dynamic, i.e. it can change in function of several traffic control policies or due to the intelligent learning capabilities of the urban traffic controller, e.g. a traffic light scheme at an intersection level may adapt its behavior to the traffic flow characteristics. Since this dynamic behavior can be difficult to model, a decentralized traffic management and control system can be perceived as a multi-agent system [4], which refers to large systems composed of interacting entities. These entities or agents can be described as *"Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed."* [5]. Vehicles and the UTCS's entities can therefore be considered under this perspective as agents. The reason to adopt this paradigm relies in the objective of having a solid basis for shifting UTMC systems from a centralized to a decentralized (or distributed in the computational sense) architecture, following known and established standards.

1.2.2 MICRO-SIMULATION

Traditional traffic simulation models consider traffic as a continuous process thereby handling it from an analytical point of view, which regards vehicle movements in aggregates [3]. These aggregates can be perceived as flows, having flow related characteristics such as average speed and density that can be accounted for analysis through fluid mechanics. This approach has however lost its popularity due to its inability to model complex, non-linear traffic operations in urban environments (e.g. complex junctions, traffic accidents, dynamically controlled traffic signals). As a consequence the results following this theory often have considerable discrepancies when compared to real life situations. On the other hand recent traffic simulation models account each traffic network user (vehicles, pedestrians, etc.), each traffic network controller (vehicle control, traffic management, sensing, etc.) and their interactions independently, hereby allowing us to model each of these element's behavior distinctively and in detail. This is referred to as a microscopic traffic simulation model, where as the prior, traditional method is regarded as a macroscopic traffic simulation model. Besides being able to implement large and complex urban networks with accuracy, other benefits of a microscopic level simulation come from the fact that such approach enables a visual representation of the problem and solution in a format which is understandable without any specialized knowledge, allowing a broader illustration of results. The micro-simulation approach allows an exact, flexible and easily understandable simulation which has the ultimate objective of forming a base for eventual policy making.

1.3 MOTIVATION

FPGAs are basically pieces of programmable logic. Being an urban traffic controller system easily perceived as an embedded system, the motivation to use FPGAs leads to numerous advantages when compared to alternatives such as ASICs or microcontrollers. When comparing to these alternatives, the major advantages in using FPGAs [8, 9] are:

- **Shorter Development Cycles** – No manufacturing steps required.

- **Field Reprogramability** – Design functionality can be upgraded which in turn prolongs product life cycles as new specifications can be implemented post development.
- **Partial Reconfiguration** – Ability to reconfigure areas of an FPGA after its initial configuration. Reconfiguration is often possible in real time, i.e. while the design is executing, allowing an uninterruptable service.
- **Processing Speed** – If the design fits in the FPGA there is no need to share the same core for multiple operations, executing various processes in parallel speeding up execution.

This choice is also supported by the continuous evolution of FPGA architectures in the last decade (and consequently drops in production costs). In figure 1.3-1, regarding Xilinx FPGAs [7], it is illustrated how FPGA usage has progressed since the mid-eighties until recent times.

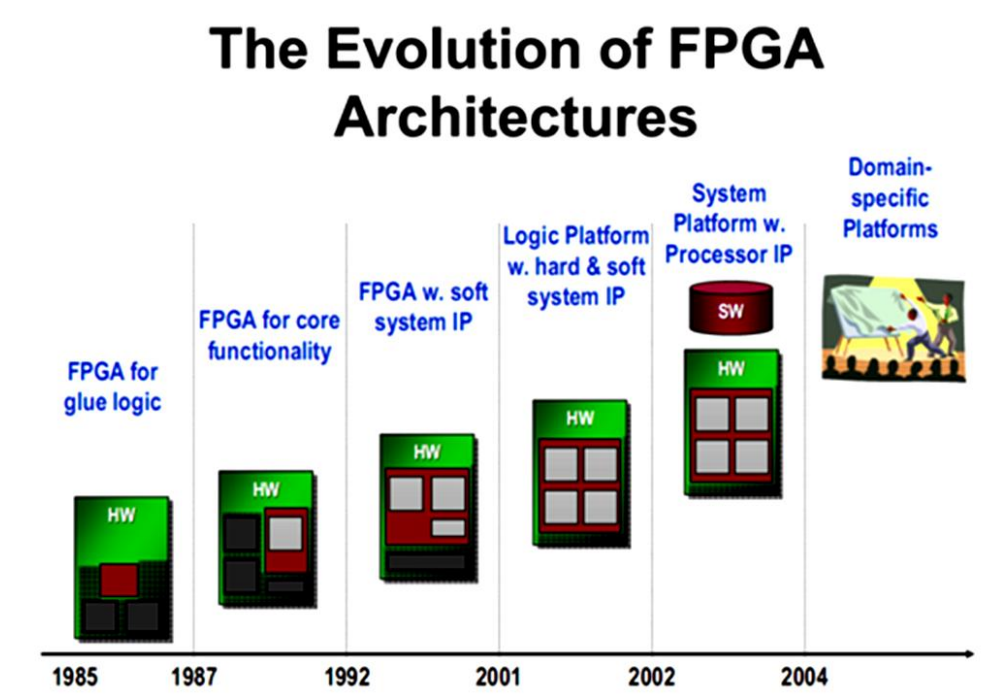


Figure 1.3-1 Evolution of FPGA architectures, figure extracted from [7], pertaining to Xilinx

As can be observed in Figure1.3-1, FPGAs are currently used as sophisticated application specific platforms and part of advanced embedded systems. It can therefore be stated that FPGAs have a positive prospect on being the dominant development platforms of the future.

Regarding performance and taking into account that in an UTMC system there are many elements which have similar behaviors (e.g. traffic lights), using FPGA technology permits us to take advantage of the fact that it offers a large-scale parallelism, by this meaning that in comparison to processor based systems as for instance general purpose computers and microcontrollers, FPGAs have the advantage of being able to handle multiple processes in parallel, hence being optimal in implementing distributed systems.

In an UTS simulation context, the scarcity of FPGA's logical hardware resources and elevated costs can limit the overall simulation's scale and features in comparison to software based implementations, the prospects are however promising since that with future advances of FPGA hardware it might be possible to implement full scale and competent simulations (in comparison to actual software based simulations) at an acceptable cost. The main focus on utilizing FPGAs as an UTMC systems simulation platform is to explore an inexistent domain in current software based simulations by allowing an effortless interface with hardware traffic controllers. Another important advantage in using FPGA hardware is that a FPGA based simulation has the potential of being many times faster than a PC equivalent, enabling designs to take part in simulations which execute in a fraction of the time necessary by software equivalents

Being UTS simulation inherently complex due to its great number of autonomous interacting entities, research is needed to aid the distributed design architecture. In response to this need the current paper, although not advancing with substantial research, leaves a reference to multi-agent systems, which lay out a solid paradigm of how to manage the cooperation between traffic entities relying on coordination mechanisms in the absence of a central control unit.

As a final motivation, the aim of the present work is to develop a hardware based UTS simulation platform tool in order to confer an initial evaluation to UTMC and/or intelligent vehicle controller designs which is inexistent in current software based simulation models. By describing a model by

means of a Hardware Description Language (HDL), and implementing this design on FPGA hardware, the result can ultimately be considered as a hardware prototype. From the applications point of view it is possible to evaluate all vehicle-traffic combinations of control, communications and interactions independently, allowing us to develop and evaluate ground-breaking concepts such as intelligent transport systems.

1.4 RELATED WORK

In terms of software UTS analysis, there are presently a large number of available tools which vary in scale of application and in many features from the ability to model incidents to the capability of taking into account weather conditions. Being constantly developed by teams of researchers or the open source community, the complexity of these tools and the focus these insist on in simulation results makes these fall out the scope of this paper. Further information and a full list of actual software based UTS analysis tools can be found by consulting “Microsimulation Tools on the WWW” (<http://www.its.leeds.ac.uk/projects/smartest/links.html>) [10].

There have been previous attempts at modeling traffic systems on FPGAs. One study from the University of Strathclyde [11, 12], Scotland, implemented a hardware description based design using Cirral and recurring to a cellular automata based model. This study obtained simulation results for a signaled four-way traffic junction, through which traffic from each lane transverse. Traffic was generated according to a circuit with a configurable creation rate. From a held simulation a graph with the average delay per car in function of several traffic light cycle times was obtained. The outcome from this simulation held similar results, in curve, to those in theoretical studies. The work also mentioning that a network calibration was in need to approximate results to real case study ones. By relying on cellular automata theory this study closely resembles software based UTS simulators, and therefore can't be directly compared to the case at study.

Another found study [13] combines software based simulations and FPGAs by dividing data sets between an FPGA and the Cray XD1 supercomputer. Although out of the context of this work it is

noteworthy that with the use of a single FPGA this execution was able to achieve a 34.4x speedup over the previously used AMD microprocessor.

It can therefore be concluded that there is currently no work directly related to the one at hand hence being an innovative concept with the possibility of uncovering a new study approach on UTSs.

1.5 THESIS OUTLINE

Chapter one presents itself as an introduction to the project and allows an understanding of the project's context by providing information about the background and purpose of this paper.

In chapter two we start by discussing all the resources needed, both hardware and software, in order to develop a similar system to the one at study.

What follows, in chapter three, is a system description starting from a broader viewpoint, at the system's top-level architecture and then proceeds with detailed descriptions of each one of the system's components. The system's component description starts off with components external to the UTS simulation per se, as are the peripheral controllers and VGA output components. The component description then passes on to UTS specifics, these include the UTS Network Infrastructure, traffic light and vehicle model implementations. We finalize this chapter by giving a detailed description of the user interface and auxiliary output components.

In a brief attempt to deliver a general idea on the current work's action and thought in the perspective of an end user, the design work-flow is presented in chapter four.

The implementation results and analysis are presented in chapter five, here we discuss the encoding process, the Dijkstra's algorithm implementation and the system in an UTS simulation perspective.

Chapter six concludes the work by gathering the most important conclusions. As a last statement a brief discussion of the applications and future work concludes this chapter and paper.

OVERVIEW

This chapter commences by describing the platform on which the design is developed and the platform on which it operates. The design work necessary for this research includes the conception of an Urban Traffic System by describing its network infrastructure and modeling the behavior of each one of its elements such as traffic light and vehicle controllers. Section 2.2 covers a summary over the development system hardware followed by a closer description of the FPGA development board and hardware peripherals required to execute and interact with the design. A brief description of each software tool and its involvement in the design process is given in sec. 2.3 before advancing with the implementation details in the subsequent chapter.

2.1 HARDWARE

The development system's setup is illustrated in fig. 2.1-1, from which can be seen that a general purpose computer is used for design flow which includes the design of the UTS model along with the FPGA related design flow (abridged as design entry, synthesis and implementation) [23]. After designing an UTS simulation model this model is then described through a Hardware Description Language (HDL), in this case VHDL[24] and after performing the remaining FPGA related design flow steps it is then implemented in the FPGA platform where it is executed.

User input is achieved by means of a PS/2 mouse and by executing the design on the FPGA platform a graphical output of the simulation and other user interaction results are accessible through a VGA monitor.

Besides being part of the FPGA related design flow, several software tools are also used for design support and verification. In the present work a general purpose computer is used in order to:

- **Develop HDL Code** – Describe the model's operation, design and organization.

- **Simulate HDL Code** – Verify the model’s description before its synthesis and implementation.
- **Design Graphics** – Enhance the user interface and VGA visualization.
- **Convert Data** – Organize information in a form that can be used by the design’s implementation.
- **Transfer Data to the FPGA Platform** – Store data in Flash Memory so it can be used during the model’s execution.
- **Program the FPGA** – The description of the model is synthesized and implemented. A program file is generated and used to transfer the design to the FPGA platform.

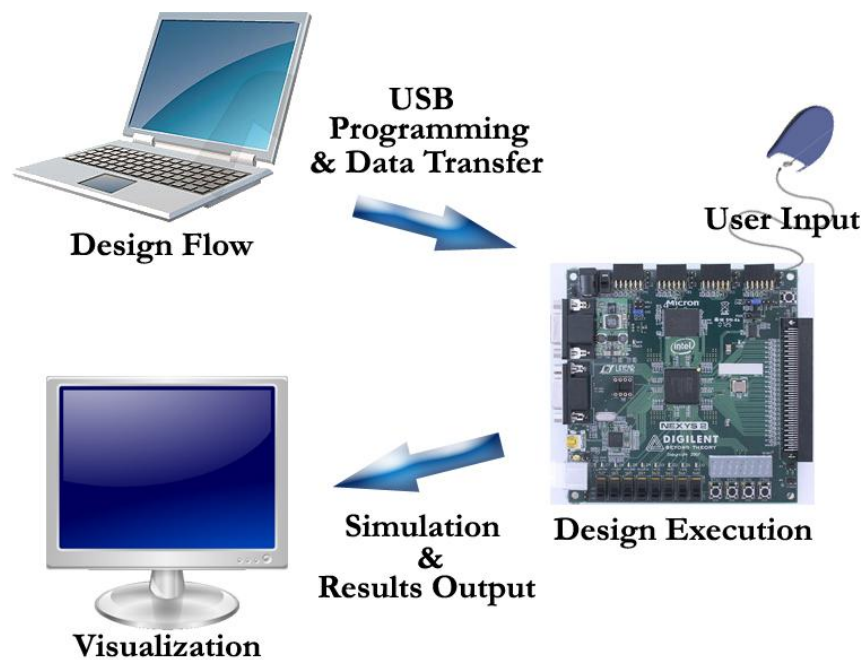


Figure 2.1-1 Development system hardware setup

For the FPGA platform the NEXYS2 development board from Digilent was chosen based on a cost versus feature comparison and availability. The NEXYS2’s main features are presented in fig.

2.1-2. This development board is built around a Xilinx Spartan 3E-500 FPGA. A comparison between various Xilinx FPGA families is presented in table 2.1-1, from where can be seen that the Spartan 3E can be considered as a low-entry product since there are much more advanced and capable FPGA chips, this aspect should be taken into consideration when analyzing the design's implementation results.

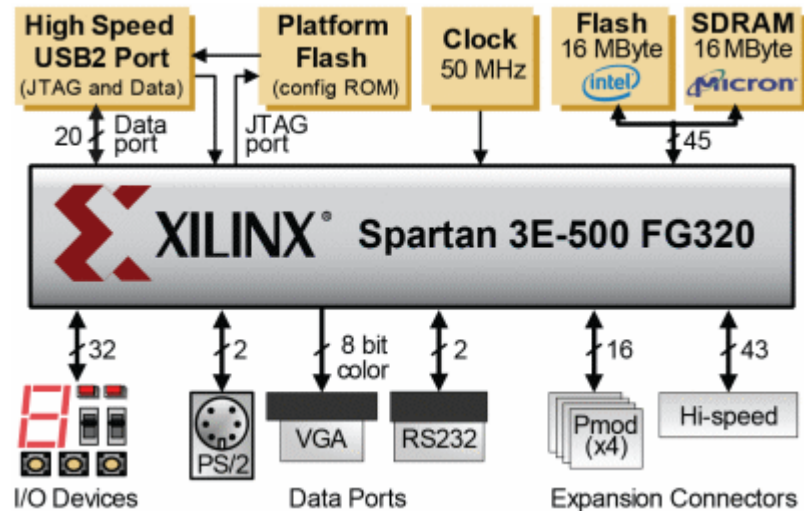


Figure 2.1-2 - Digilent Nexys2 Board features overview, figure extracted from the Nexys2 Reference Manual available at the Digilent website

Features	Virtex-6	Virtex-5	Virtex-4	Extended Spartan-3A	Spartan-3E	Spartan-3E 500
Logic Cells	Up to 758,784	Up to 330,000	Up to 200,000	Up to 53,000	Up to 33,192	10,476
User I/Os	Up to 1200	Up to 1200	Up to 960	Up to 519	Up to 376	232
Clock Management - DCM	Yes	Yes	Yes	Yes	Yes	Yes
Embedded Block RAM	Up to 26 Mbits	Up to 18 Mbits	Up to 11Mbits	Up to 1.8 Mbits	Up to 648Kbits	360Kbits
Soft Processor Support	Yes	Yes	Yes	Yes	Yes	Yes
Embedded PowerPC® Processors	-	Yes (PowerPC 440 Processor)	Yes (PowerPC 405 Processor)	No	No	No

Table 2-1 – Xilinx FPGA Family comparison including Spartan 3E-500 specifics

Besides giving use to the PS/2 input, VGA output and USB (file transfer and programming) related features stated above, the present work gives use of such features of the NEXYS2 development board as the FLASH memory for graphic data storage (in bitmap form) and a push-button for a user-reset.

In terms of programming resources and the conceptual structure of a FPGA device, an overview of the Xilinx Spartan-3 FPGA Family can be obtained by consulting Pong P. Chu. *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version* (11-13) [14]. It is however important to mention memory storage in FPGAs. Xilinx Spartan-3 devices are made up of logic cells and macro cells. There are two methods to store information on an FPGA chip. One is to use the lookup tables present in logic cells, which are normally used for logical functions. By combining many logic cells a RAM element can be formed, which is denominated in this case by distributed RAM. On the other hand the FPGA contains four types of macro block one of which is a block RAM, which is an 18k-bit synchronous SRAM that can be arranged in various types of configurations, from single port to dual port memories. The main difference between these two types of memory storage is that distributed memory has ideally zero latency and can be addressed by multiple circuits while block RAM based memory components are limited to two ports and have read and write latencies. Distributed memory is however avoided when possible as it occupies a great deal of logic resources that could otherwise be used to implement logic functions.

2.2 SOFTWARE

Xilinx ISE 10.1 & ModelSim XE III 6.3c

Regarding the FPGA related design flow [23], in order to transfer the design to the FPGA, it is first necessary to perform design entry by resourcing to a hardware description language, in this paper this entry was done with VHDL [14, 24], another possibility is Verilog. Before performing synthesis and implementation an optional step is software based simulation as to assure the functionality of the design at hand, this is often done for debugging or since synthesis and implementation can take an extensive amount of time to complete. The present work uses Xilinx ModelSim XE III 6.3c for simulation. After simulating the design the next steps are synthesis and

implementation, where the hardware description is translated to a programming file which contains all the information needed to implement the design in the FPGA platform. Design entry and the remaining FPGA design flow were performed using Xilinx ISE 10.1. Both software packages can be obtained in Xilinx's website, and a closer understanding in the design flow can be obtained by consulting *ISE Help: FPGA Design Flow Overview* http://www.xilinx.com/itp/xilinx8/help/iseguide/html/ise_fpga_design_flow_overview.htm.

Digilent Adept Software Suite & MemUtil

After generating a programming file it is necessary to program the FPGA. This was done through the Digilent ExPort utility, a software utility part of Digilent Adept Software Suite.

In order to transfer data to the NEXYS2 development board's flash memory via the PC-FPGA USB interface, it is necessary to resort to Digilent MemUtil.

Digilent MemUtil and the Digilent Adept Software Suite can be obtained through Digilent's website (www.digilentinc.com).

Mathworks MATLAB

Matlab is a powerful general purpose mathematical calculation and analysis tool. In this work MATLAB is used to create functions which convert text and graphics data to a form which is usable by the VHDL description and the FPGA platform. Matlab is also used to evaluate results such as those from the shortest path finding algorithm [15] implementation discussed further on in chapter three with the user controlled vehicle model.

Others

Other tools were used to support or facilitate design and development. These tools include Adobe Photoshop which was used to design the traffic network's graphical representation and user interface menus. Another tool used in this work is Tiled, a tile-map editor which speeds up the design process of the traffic network's graphical representation.

3 DETAILED IMPLEMENTATION

OVERVIEW

In this chapter descriptions are given in close relation to their VHDL entries. When possible we first present an upper level abstraction followed by a detailed descriptions of each component. Following this method this chapter commences by presenting the top-level system's architecture in section 3.1 with a brief description of each one of its composing elements. We then advance with a detailed description of each of the system's components, starting with the peripheral controllers in section 3.2. The peripheral controllers relate to the system's input, output and other basic functionalities, and include the VGA, PS/2 and FLASH memory controllers. The UTS network's infrastructure is presented in section 3.3. The UTS network's infrastructure allows other network elements to situate themselves in the traffic network and therefore serves as a road map. Next up in section 3.4 we discuss the traffic light controllers, where a basic as well as an intelligent traffic light models are described. In sec. 3.5 the implementation of a basic vehicle model as well as a user controlled vehicle with a shortest path calculation feature is presented. The user interface module is discussed in section 3.6. Since a desired characteristic in the UTS simulation is a visual output there is the need to implement auxiliary output components that support the graphical output for both traffic lights and vehicles, these output components are discussed lastly in sec. 3.7.

3.1 ARCHITECTURE: TOP LEVEL

Figure 3.1-1 illustrates an overview of the system and its principal components or modules.

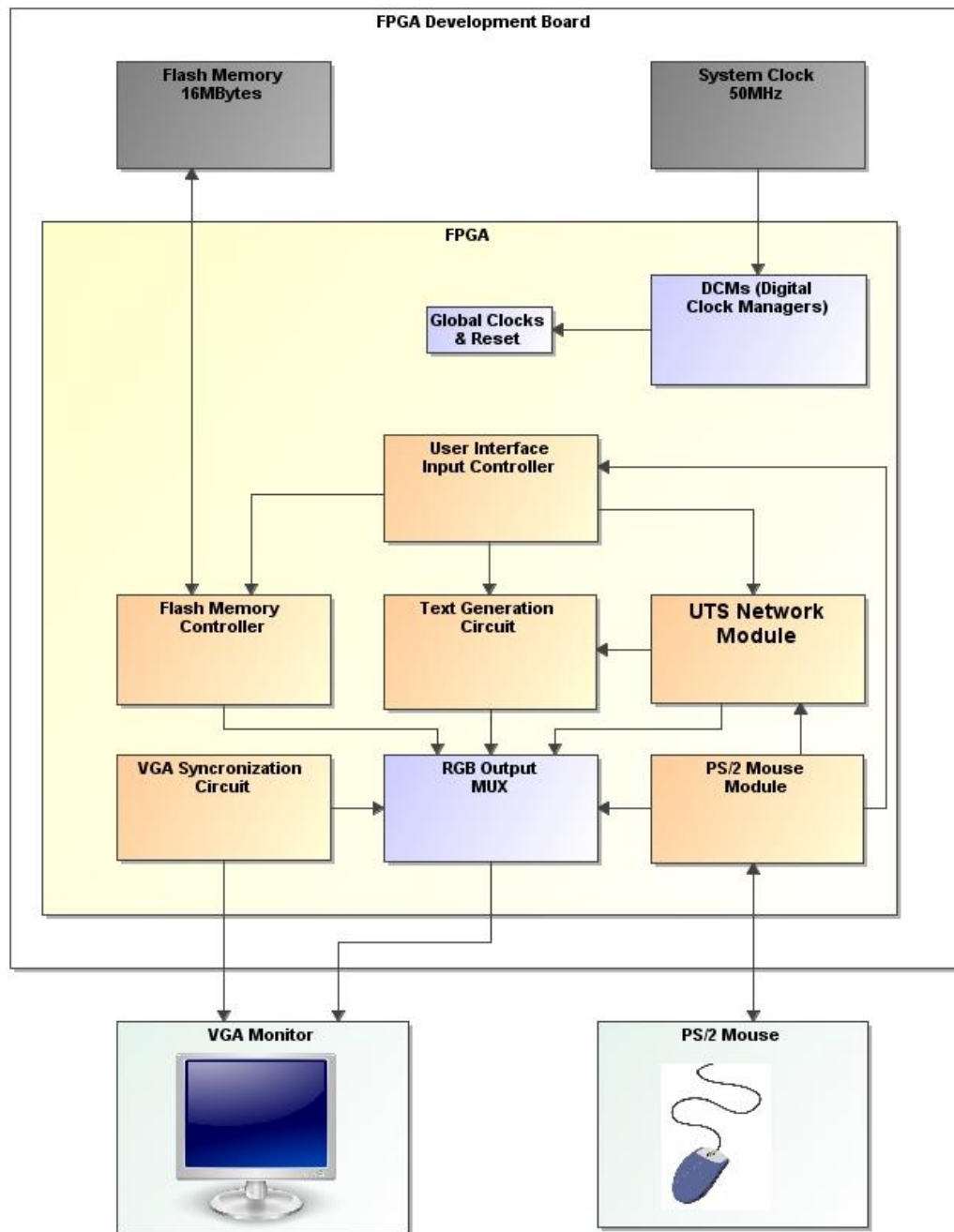


Figure 3.1-1 Top Level System Architecture

The Spartan-3E FPGA incorporates a total of four digital clock managers (DCMs) which allow an extensive manipulation and conditioning of the system clock, a complete description of these can be

obtained by consulting the Spartan-3 Generation FPGA user guide (UG331) available at Xilinx's website (http://www.xilinx.com/support/documentation/user_guides/ug331.pdf). In the current work 50MHz, 25MHz and 12.5MHz clock signals are generated from DCMs (fig. 3.1-2) and distributed automatically, during synthesis, throughout the design through clock buffers. The reason behind using different clock frequencies is based on the fact that some circuits benefit from these clock signals for timing issues (e.g. VGA synchronization and flash memory access), while other circuits benefit from a less intense physical synthesis due to “loosened” timing requirements and consequent reducing of logical resources usage, as often techniques such as logic replication are in need to fulfill timing requirements.

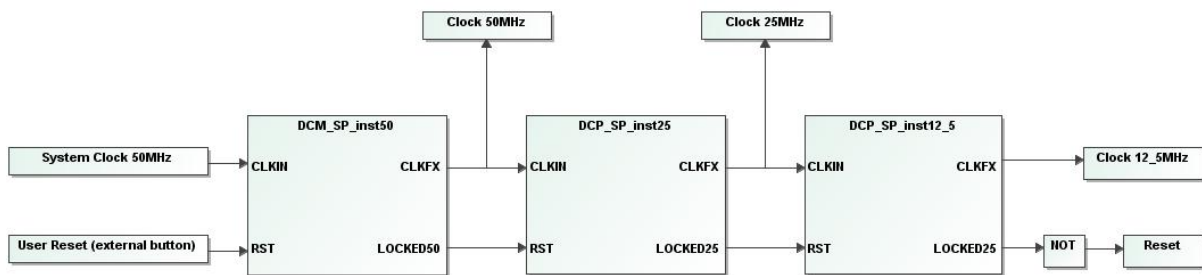


Figure 3.1-2 System clock signals generation with DCMs, here we generate 50Mhz, 25Mhz and 12.5Mhz clock signals

The peripheral controllers provide input, output and storage to the design, and are comprised by the PS/2 mouse module, VGA synchronization and RGB multiplexer circuits and flash memory controller respectively. A text generation circuit and a user interface module were designed in order to support user input and simulation data output. The main simulation event occurs in the UTS Network Module which incorporates the system's core in terms of functionality. A detailed explanation of each component is given in the following sections.

3.2 PERIPHERAL CONTROLLERS

3.2.1 VGA CONTROLLER

This section describes the VGA Synchronization circuit which is necessary to provide compatible timing signals in order to drive a VGA compatible display.

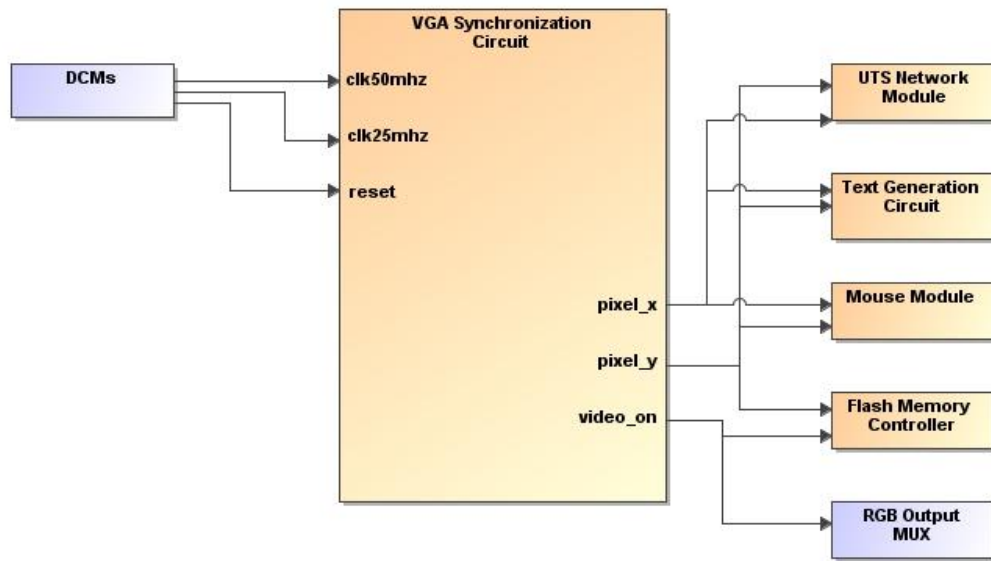


Figure 3.2.1-1 VGA Synchronization Circuit

The VGA Synchronization circuit (symbolized in fig. 3.2.1-1) generates timing signals that define a 640-by-480 pixel resolution with a 50Hz refresh rate. Besides generating the timing signals necessary to drive a VGA monitor the VGA synchronization circuit also provides the pixel coordinates to other modules, which are necessary be it primarily for data output in a graphical form or for user data input in conjunction with the PS/2 mouse cursor coordinates. A full explanation on VGA synchronization and an identical VHDL description to the one used in this work can be found by consulting Pong P. Chu. *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version* (260-267) [14].

3.2.2 PS/2 MOUSE MODULE

Figure 3.2.2-1 illustrates the mouse module's architecture. The mouse controller is comprised of a PS/2 transmission and reception circuit which handles the communication via the PS/2 port of the development board. The Position Coordinates and Control finite-state-machine (FSM) uses the transmission and reception unit "PS2_rxtx_unit" and communicates with the PS/2 mouse setting it up in stream mode in which the mouse transmits data packets when movement occurs or when the states of the mouse buttons change.

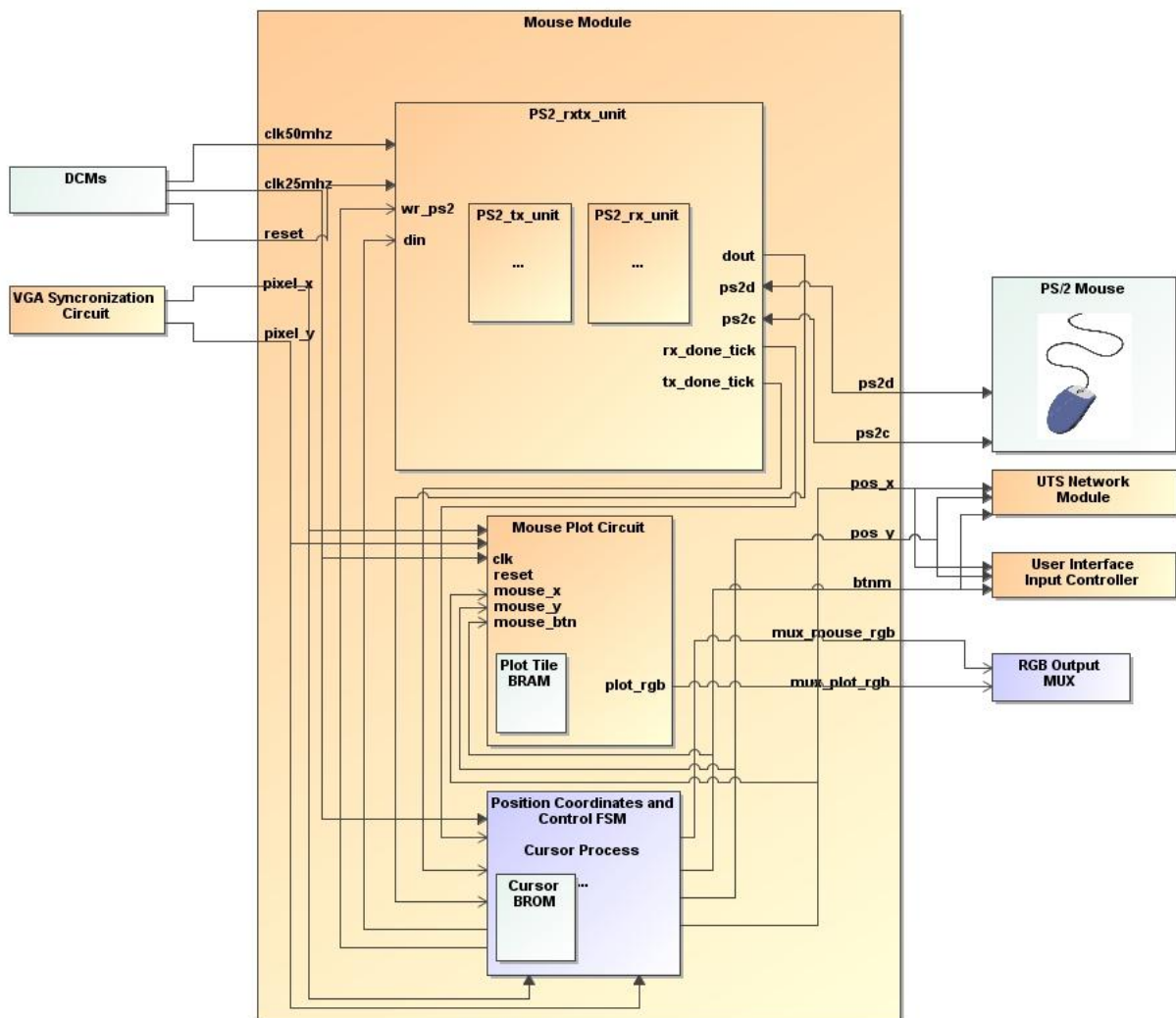


Figure 3.2.2-1 Mouse Module Architecture

The mouse cursor graphics are stored in a 16 Bit wide by 22 deep Block RAM based ROM component (contents are shown in VHDL Description 3.2.2-1) which is addressed according to the current pixel coordinates and whose output is redirected to the RGB multiplexer circuit that prioritizes the RGB signals to output. The circuit operation is optimized for low FPGA resource usage. In regard to the operation the circuit verifies if the current pixel coordinates are inside the cursor boundaries by subtracting these with the mouse coordinates. If the vertical pixel coordinate component falls out of the vertical cursor boundary then the ROM's last word is addressed which is a null word. The mouse RGB signal is defaulted to "000" inside the process, and is only assigned to a different value if the horizontal pixel coordinate falls inside the horizontal cursor boundary and the corresponding ROM output bit (which is accessed by subtracting the pixel and mouse horizontal coordinates and utilizing the result as an array index) has the value '1', in this case the cursor RGB value is assigned according to the mouse button's state through configuration parameters this way giving the user feedback.

```
; 16-bit wide by 22 deep ROM
memory_initialization_radix = 2;
memory_initialization_vector =
1000000000000000,
1100000000000000,
1110000000000000,
1111000000000000,
1111100000000000,
1111110000000000,
1111111000000000,
1111111100000000,
1111111110000000,
1111111111000000,
1111111111100000,
1111111111110000,
1111111111111000,
1111111111111100,
1111111111111110,
1110011111111111,
1100011111111111,
1000001111111111,
0000001111111111,
0000000111111111,
0000000011111111,
0000000001111111,
0000000000111111,
0000000000011111,
0000000000001111,
0000000000000111,
0000000000000011,
0000000000000001,
0000000000000000;
```

VHDL Description 3.2.2-1 – Cursor ROM contents

An additional plotting circuit was designed which aids the user interface experience by providing a simple screen drawing mechanism (similar to Microsoft Paint, albeit a rudimentary implementation) which is always accessible and that can be useful to mark zones in the screen where the user makes real-time alterations to simulation parameters, so the user can recall these later. This circuit divides the screen in 16-by-16 pixel tiles, and stores a bit value for each tile in a BRAM component. When the user clicks on the screen with the right-mouse button this tile is written as occupied, on the other hand when the user clicks on the screen with both the right and left-mouse buttons the tile is written as unoccupied clearing the screen in the correspondent position. According to the current pixel coordinates being drawn the RAM component is accessed and its output is encoded in a configurable RGB signal value which connects to the RGB multiplexer circuit.

The mouse coordinates and the button states are useful to other modules such as the user input and UTS network modules therefore these signals are connected to these modules.

The bidirectional PS/2 communications controller used in this work is based on the circuit description found in Pong P. Chu. *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version* (200-214) [14].

3.2.3 FLASH MEMORY CONTROLLER

The flash controller is symbolized in figure 3.2.3-1. The *video_on* and *pixel_y* signals are provided in order to synchronize memory access since bitmap information is fed in real-time to the RGB output multiplexing circuit. A memory offset value is provided by the user interface input controller which basically indicates the flash controller which of the stored images it should draw in the screen.

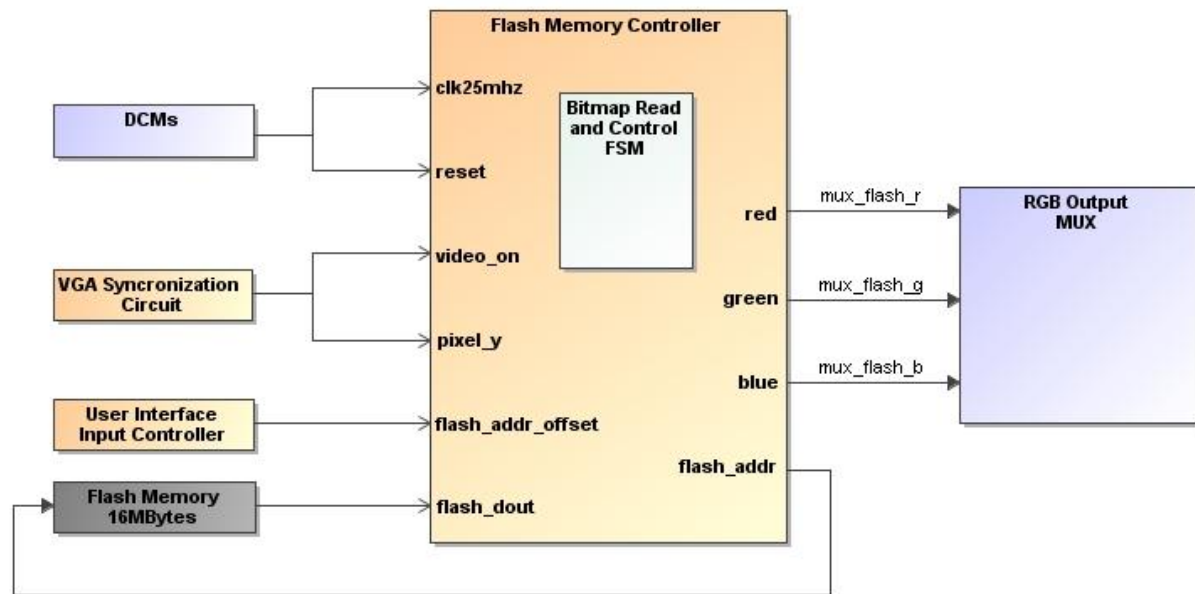


Figure 3.2.3-1 Flash memory controller

The flash video memory controller used in this work is based on the controller found in Richard E. Haskell, Darrin M. Hanna. *Learning By Example Using VHDL, Advanced Digital Design With a NEXYS2 FPGA Board* (182) [15] which is originally designed to draw a single bitmap image on the screen. This functionality was enhanced with the capability of addressing multiple memory regions by introducing an offset value (discussed further on) in the flash memory read address. The present work uses flash memory uniquely as bitmap data storage, as this would very quickly exceed the FPGA's logical resources if it were to be stored in such. The stored bitmap data represents the UTS network's infrastructure as well as the graphical user interface. The user interface is mainly based in an interaction between the mouse and graphical user interface which resumes to menus composed of graphical buttons. By knowing a button's graphical region (boundaries) and the mouse cursor's position in terms of display pixel coordinates, an interaction is implemented by comparing both to evaluate whether or not the mouse cursor is situated inside this graphical region which as mentioned above represents a button in the user interface. The user interface is discussed in detail in section 3.6 of the current chapter.

Since the NEXYS2 uses 3 bits to code each of the red and green color signals and 2 bits to code the blue color signal, each pixel is then represented by a single byte. This method of storing image information is known as 8-bit truecolor. Because computer drawn bitmaps are normally coded in 16bit or 32 bit formats, and other image format's such as jpeg are often used, it is therefore necessary to convert an image to the bitmap format compatible with the NEXYS2 VGA output before transferring it to the flash memory. Although a suggested Matlab function to perform this conversion is presented in Richard E. Haskell, Darrin M. Hanna. *Learning By Example Using VHDL, Advanced Digital Design With a NEXYS2 FPGA Board* (176) [15], the processing time required for this function to complete the conversion seems to be unnecessarily long (in excess of ten minutes on a 1.73Ghz Intel Core Duo processor). In an attempt to optimize the conversion time the following (Figure 3.2.3-2) conversion function was coded instead, which takes less than a minute in comparison with the previous function to perform the same conversion. The function outputs a binary file which can then be transferred to the flash memory by programming the FPGA with a specific program file that enables flash memory programming operations by executing on the PC side Digilent's MemUtil software tool and transferring information via USB. The FPGA programming file can be downloaded from Digilent's website. (<http://www.digilentinc.com/Data/Products/NEXYS2/Nexys%20%20500K%20BIST.zip>)

```

1  function [img2,OutVector]= IMG2ExtMem(imgfile, outfile)
2  -   img = imread(imgfile);
3  -   height=size(img, 1);
4  -   width=size(img,2);
5  -   s=fopen(outfile,'wb');
6
7  -   img2Out(:,:,1) = img(:,:,1)./32;
8  -   img2Out(:,:,2) = img(:,:,2)./32;
9  -   img2Out(:,:,3) = img(:,:,3)./64;
10 -   OutMatrix=img2Out(:,:,1).*32 + img2Out(:,:,2).*4 + img2Out(:,:,3);
11 -   OutMatrixT=OutMatrix';
12 -   OutVector=OutMatrixT(:);
13 -   img2(:,:,1) = img2Out(:,:,1).*32;
14 -   img2(:,:,2) = img2Out(:,:,2).*32;
15 -   img2(:,:,3) = img2Out(:,:,3).*64;
16
17 -   fwrite(s, OutVector, '307200*uint8');
18
19 -   fclose(s);

```

Figure 3.2.3-2 Image to FLASH bitmap conversion function

This conversion has an obvious negative visual impact on the image being converted since it reduces the number of colors of the converted image to 256; this however is not a major concern as graphical quality isn't being focused on this work.

Having interest in storing more than one bitmap due to the user interface and in regard to the initially mentioned offset mechanism, considering that the screen resolution is 640-by-480 Pixels, it is obvious that this should be the size of each bitmap. Taking into account that each pixel is represented by a single byte this implies that a 640-by-480 pixel bitmap occupies a total of 307200 Bytes. Since the flash memory output is 16-bits wide, the bitmap occupies a corresponding total of 153600 memory words. Regarding the possibility that in a future development each bitmap may have additional stored information related to it (default parameters, additional text information, etc.), this work considers separating the FLASH memory in 200000 word memory segment blocks, (each memory block providing enough space to store a 640-by-480 Pixel bitmap and a reserved memory space for future use). Therefore the offset value between each memory block is correspondent to the size of each block, or 200000 as illustrated in figure 3.2.3-3, in which each bitmap is read by providing its corresponding block number through the *flash_addr_offset* signal, which is indicated by the user interface input controller (see figure 3.2.3-1).

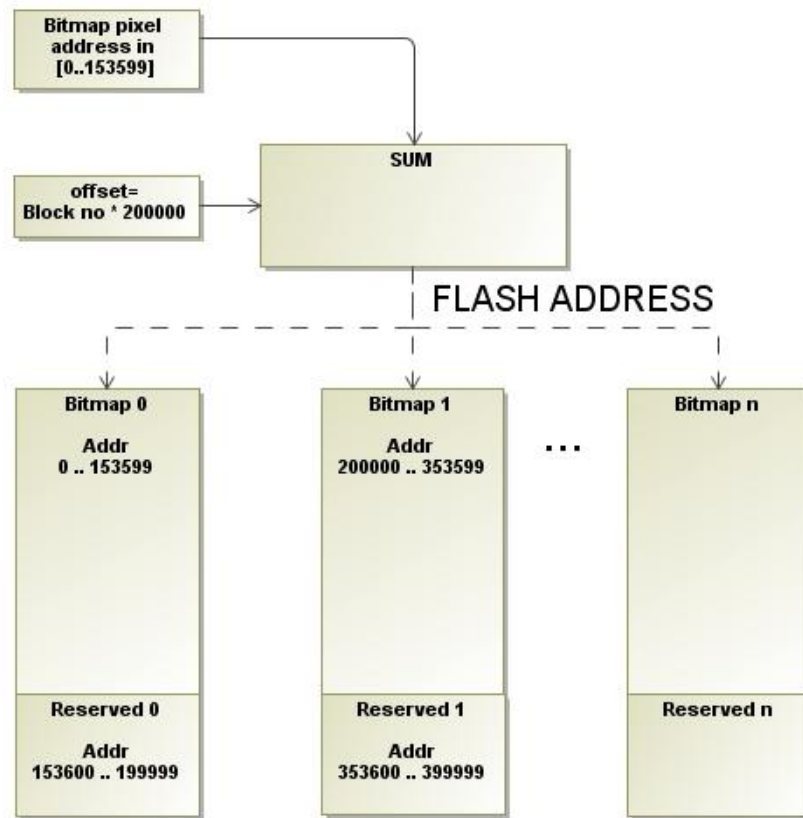


Figure 3.2.3-3 FLASH memory addressing with offset mechanism

The NEXYS2 board has a 16-MBytes flash memory, since we separate this memory into 200000 16-Bit word blocks this memory component is capable of storing a total of forty bitmap blocks using the previous scheme.

Since we are discussing graphics it seems adequate to include the methodology involved in designing a graphical representation of the UTS network's infrastructure (or road map). It is therefore necessary to draw an image file which represents the UTS's network. The present work only considers UTS networks whose routes follow the vertical or horizontal coordinate axis. This choice was made through trial and error where it was concluded that other schemes could lead to an overwhelming complexity due to numerous factors, mainly related to vehicle navigation and graphical output. Therefore only up to four-way traffic junctions were considered as to avoid complexity overheads, so each intersection can incorporate an East, North, West and South routes.

In order to facilitate and speed up the drawing of an UTS network road map a tile-map editor (Tiled) was used to draw the network. This program can be obtained in <http://mapeditor.org/downloads.html>, and further information can be obtained by consulting the included documentation. A costume 16-by-16 Pixel tile tile-map (see figure 3.2.3-4) was designed to support the network drawing. A screenshot of the tile-map editor being executed can be seen in figure 3.2.3-5, and in figure 3.2.3-6 a resulting graphical representation of an UTS network used for simulation is presented.

Having drawn an UTS network infrastructure the next step, following the suggested design flow (Chapter 4), is to number each intersection as to identify it for further reference, this has also been done as can be observed in figure 3.2.3-6.

Since the original image has few colors to start with the conversion's color downgrade aren't expected to be noticeable.



Figure 3.2.3-4 Tile map used to draw the network road map, each tile is 16-by-16 pixel



Figure 3.2.3-5 Tiled: tile-map editor being executed, provides a fast and easy placement of tiles on a canvas



Figure 3.2.3-6 Graphical representation of an UTS network road map where the intersections have been numbered for later identification

3.3 UTS NETWORK

3.3.1 ARCHITECTURE

Since the UTS Network Module is rather complex when compared to other modules (in fig. 3.1-1) it may be beneficial to discuss this module's architecture preceding the discussion of its composing elements. The UTS Network module's architecture is illustrated in figure 3.3.1-1 and a brief description of each of its elements:

- **U0_Timer:** Sub-module which performs a clock division and delivers 1, 0.1, 0.03 and 0.01 second tick signals. These are useful to other elements as for example the traffic light controller which updates the traffic lights status and timers when the second tick signal is asserted.
- **U1_Intpos_BRAM:** Dual port read-only memory implemented in block ram and which holds the intersection position coordinates, therefore being part of the network infrastructure data set. This information is accessed by vehicles that request such information when updating their graphical position.
- **U2_Timer_BRAM:** Random access memory which holds the running timer values for each traffic light. The traffic light controller is responsible for updating these.
- **U3_Timeouts_BRAM:** Random access memory which holds the timeout values associated with each traffic light status, which is used for comparison with the timer values by the traffic light controller.
- **U4_LFSR:** Sub-module that generates a pseudo-random two bit signal which is used by the “dummy” vehicle model to implement random route taking decisions by vehicles.
- **U5_Vehpos_BCAM:** Content-addressable memory which integrates a VGA output circuit. This component and the U6_Intpos_BCAM are discussed in section 3.7.
- **U7_Dij_prev_BRAM:** Random-access memory that holds the results of the Dijkstra’s shortest path finding algorithm. These results are utilized by the user controlled vehicle in order to take the shortest path towards a destination intersection.
- **U8_Dij_dist_BRAM:** Random-access memory that holds the distance values of each intersection to the source intersection. These result from executing the shortest path algorithm.
- **Traffic Lights FSM:** Composes the traffic light model.

- **“Dummy” Vehicle FSM:** Composes a random route taking vehicle model.
- **User Controlled Vehicle FSM:** Composes a shortest route taking vehicle model.
- **Dijkstra’s Algorithm FSM:** Finite-state-machine based implementation of the Dijkstra Shortest Path Finding (SPF) algorithm.
- **User Controlled Vehicle:** User Input FSM: Obtains the target intersection through PS/2 Mouse user input and coordinates the operations of the User Controlled Vehicle FSM and the Dijkstra’s Algorithm FSM.
- **Clock MUX:** Supports the user controlled vehicle debugging by allowing the user to manually control the clock signal of the user controlled vehicle related controllers.

Output from the UTS Network Module is either graphical data or debugging results. In regard to inputs besides the obvious signals the user interface input originated signals allow the user to alter simulation related parameters in real time. User controlled clock signals are also provided serving the basis for debugging and mouse inputs are to provide the user control over the user controlled vehicle.

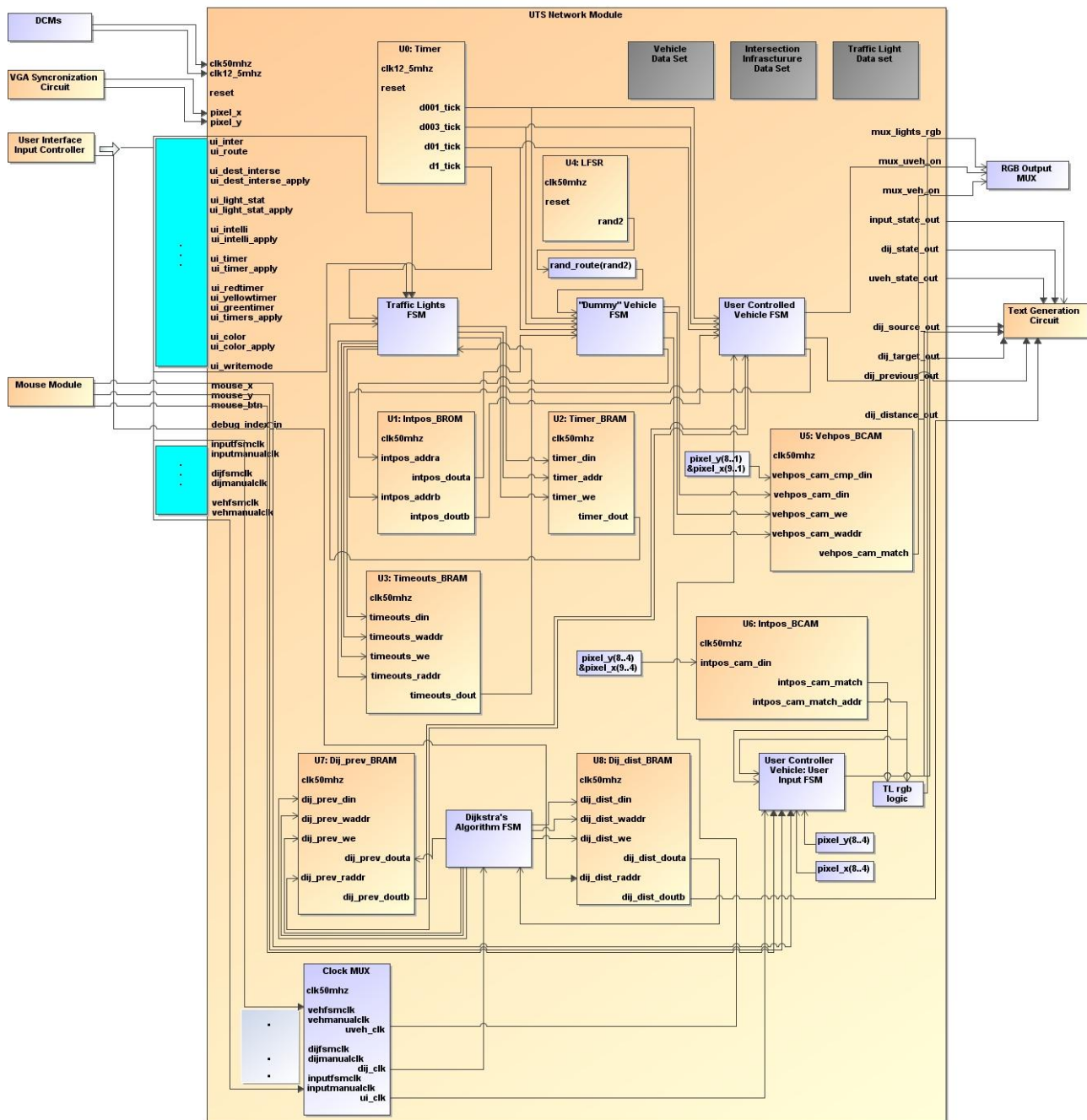


Figure 3.3.1-1 UTS Network Module Architecture

3.3.2 NETWORK INFRASTRUCTURE DATA SET

Having arbitrated a graphical representation for the UTS network's infrastructure, it is necessary to write a corresponding VHDL description which is capable of holding a characterization of this or any other UTS network road map drawn following the same procedures, as well as providing

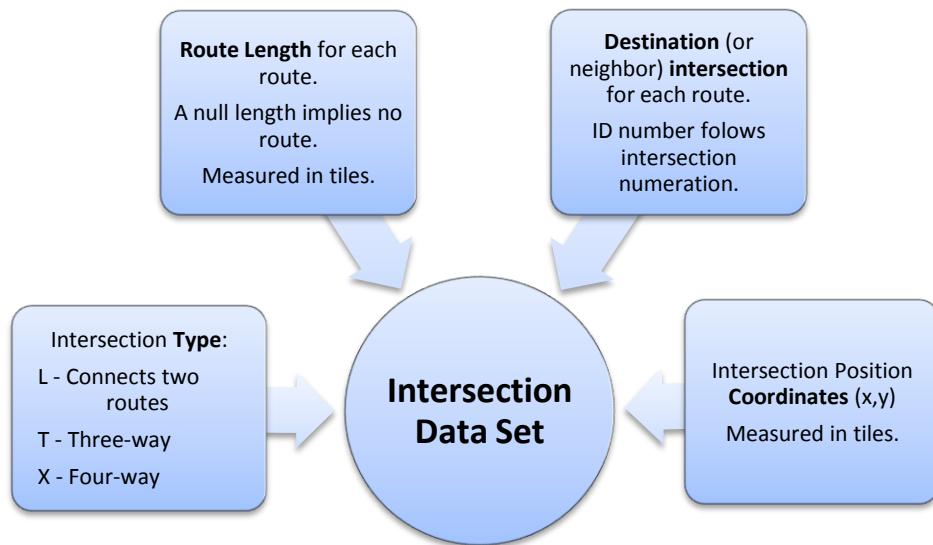


Figure 3.3.2-1 UTS network infrastructure data set

sufficient information for traffic management controllers (traffic lights) and network users (vehicles) to situate themselves or navigate through the network. The identified data elements necessary to hold this description in reference to a single intersection are illustrated in figure 3.3.2-1. In order to being able to completely describe the network's structure in a decentralized manner the descriptions for each of the UTS's composing intersections are aggregated in the same order as they were numbered, this provides a simple mechanism in which each intersection can be addressed according to its identification. The reference each intersection has to its neighbor intersections is similar to the implementation of linked lists in programming languages such as C.

A description of each intersection data set element is given next:

- **Intersection Type:** Since the present work considers up to four-way intersections, an identifier regarding the number of routes is useful (e.g. there is no sense in a vehicle respecting the right-hand priority rule in a two-route intersection). Since this work discards a “one route intersection” also known as a dead-end street, there remains three types of intersections in relation to the number of routes the intersection comprises. The mnemonics chosen to represent these are “L”, “T”, and “X”, referring to a two (Link), three and four route intersection respectively.
- **Route Length:** Assuming a vehicle keeps track of the distance it has traveled in its current route the vehicle can compare this value with the route length with the objective of knowing if it has reached the end of the route. The route’s length is also useful in shortest path calculations. Although the route’s length can be calculated by subtracting the position coordinates of adjacent intersections, the present work considers doing this calculation before-hand since this value is considered not to change during execution. Each intersection holds the route length for each route (East, North, West and South). This introduces a redundancy since two adjacent intersections will have the same route length value for the route that connects both. Due to this fact route length for a non-existent route must be considered to be zero.
- **Destination Intersection ID:** As mentioned earlier, each intersection is numbered as to provide a means of addressing it similarly to a linked list. Therefore each route of the intersection must have an intersection identification number, which should be zero when the route is non-existent, or when a one-way route is desired which supports the traffic infrastructure as a bi-directional graph. This could also be implemented by specifying the route length as zero, but this would difficult real-time editing by the user since it would be more difficult to keep track of the route’s distance value than the neighbor route identification number.

- **Position Coordinates:** Without position coordinates is difficult or even impossible to correlate the UTS infrastructure's description with its graphical representation. This is important so that traffic controllers pertaining to a certain intersection or network users such as vehicles can output their positions correctly if necessary.

As an example of how to translate a graphical representation to the corresponding VHDL description figure 3.3.2-2 represents the situation where the first intersection (intersection number one) of the previously presented UTS network (in fig. 3.2.2-6) is to be described in VHDL. It is important to state that the bitmap has been divided in 16-by-16 pixel tiles which then compose the position coordinates instead of single pixels as to simplify the implementation, increasing the scale also lowers resource usage by limiting the registers size which hold the position coordinates and route length values. As mentioned earlier intersection position coordinates are important so that for example vehicles, knowing their origin intersection and current route, can update their position on the screen correctly. Other coordinate scales are possible by altering the tile size and the corresponding route lengths. In this case a route length unit is exactly one tile, or sixteen pixels. It is of importance to note that this work doesn't focus on a calibration relative to dimensions as to approximate the implementation to a real case scenario.



Figure 3.3.2-2 Intersection describing example

Taking up on intersection one's description, its coordinates can be found by inspecting figure 3.3.2-2, by which they are observed to be $(x, y) = (2, 0)$. Intersection one's type is clearly L (Link) since it only connects two routes. By observing the figure it is seen that there are no routes to the east or north from this intersection, therefore the route length and destiny intersection for these route directions must be zero. It can also be noticed that there is a distance of seven tile squares between the centers of intersection one and intersection two. This infers that the route length for the west route is seven and the destiny intersection is intersection two. Repeating this procedure for the south route we arrive to the conclusion that it has a length of ten and the destiny intersection for this route is intersection ten.

VHDL Description

The VHDL data structure to contain the intersection's data set elements can be implemented using VHDL record data types defined as following (VHDL Description 3.3.2-1):

```

1  constant MAX_ROUTE_LENGTH  : natural := 32; -- Unit is 16 (pixels).
2  constant MAX_INTERSECTIONS  : natural := 50; -- Stipulated limit of intersections
3  type ROUTES_ENUM            is (E, N, W, S); -- East, North, West and South.
4  type INTER_TYPE              is (X, T, L); -- Intersection type: X, T and Link.
5  type ROUTE_LENGTH_TYPE      is array(ROUTES_ENUM'left to ROUTES_ENUM'right)
    of natural range 0 to MAX_ROUTE_LENGTH-1; --NULL means no route attached.
6  type  NEIGHBOR_INTER_ID_TYPE      is  array(ROUTES_ENUM'left      to
    ROUTES_ENUM'right) of natural range 0 to MAX_INTERSECTIONS;
7  type INTER_REC_CONST is RECORD
8    inter_type      : INTER_TYPE;
9    route_length    : ROUTE_LENGTH_TYPE;
10   neighbor_inter : NEIGHBOR_INTER_ID_TYPE;
11 end RECORD;
12 type  INTER_ARRAY_CONST_TYPE  is  array(1  to  MAX_INTERSECTIONS) of
    INTER_REC_CONST;

```

VHDL Description 3.3.2-1 Intersection data structure

Having multiple intersections, a decentralized description of the network's infrastructure can therefore be obtained by creating an array composed by each one of the intersection's descriptions. As a practical example, the previously designed UTS network (represented in figure 3.2.2-6) hereby has the following description (VHDL Description 3.2.2-2) starting with intersection one towards intersection fifty, in which only up to intersection five is represented. We can observe that the first intersection's description follows the previous example were we described intersection one.

```

1 signal inter_s_const : INTER_ARRAY_CONST_TYPE := (
2 (inter_type => L, route_length => (E=>00, N=>00, W=>07, S=>05), neighbor_inter =>
   (E=>00, N=>00, W=>02, S=>10)),
3 (inter_type => T, route_length => (E=>07, N=>00, W=>02, S=>05), neighbor_inter =>
   (E=>01, N=>00, W=>03, S=>11)),
4 (inter_type => T, route_length => (E=>02, N=>00, W=>12, S=>03), neighbor_inter =>
   (E=>02, N=>00, W=>04, S=>07)),
5 (inter_type => T, route_length => (E=>12, N=>00, W=>05, S=>03), neighbor_inter =>
   (E=>03, N=>00, W=>05, S=>09)),
6 (inter_type => T, route_length => (E=>05, N=>00, W=>06, S=>06), neighbor_inter =>
   (E=>04, N=>00, W=>06, S=>17)),--5
7 ...
8 );

```

VHDL Description 3.3.2-2 UTS Network Infrastructure description
example of first five intersections

In the previous description we didn't include one of the identified network infrastructure elements, which is the intersection's position coordinates. In order to spare resource usage, since the previous infrastructure elements were stored in distributed ram, the intersection's position coordinates are stored in a 11-bit wide by fifty deep read-only memory (see figure 3.3.2-3). The first five bits of the memory word corresponds to the vertical tile coordinate and the last six bits to the horizontal tile coordinate. The ROM component's interface (and further memory components) was implemented using the Block Memory Generator IP Core accessible via the IP Coregen utility integrated into Xilinx ISE. More information on using Block RAM with Spartan3E FPGAs can be found by consulting *Using Block RAM in Spartan-3 Generation FPGAs*. Xilinx, XAPP463 [19]. More information about the Block Memory Generator IP Core can be obtained by consulting *Block Memory Generator V2.8, DS512*. Xilinx [18]. This memory component uses Block RAM macro blocks and is accessed,

as will be discussed further on, by the vehicle controllers. The resulting memory component has a single port and one clock cycle memory read latency, therefore only one vehicle can access it at a time and the latency has to be taken into account when designing the vehicle's controller. By using Block RAM a considerable amount of logical resources are spared comparing to the situation where the intersections positions are stored in distributed memory. Being the memory component read-only, its contents must be specified before design synthesis, therefore it is necessary to create a "coe" (coefficients) initialization file to hold these contents. The first five intersection coordinate values in the ROM initialization file for the previously designed UTS Network are described in the following VHDL Description 3.3.2-3.

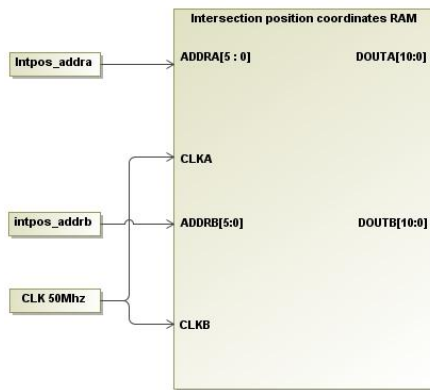


Figure 3.3.2-3 Intersection position coordinates memory component

```

; 11-bit wide by 50 deep ROM
memory_initialization_radix = 2;
memory_initialization_vector =

;2345123456
00000000010,
00000001001,
00000001011,
00000001011,
00000010111,
00000011100,
...

```

VHDL Description 3.3.2-3 Intersection position coordinates RAM
initialization file's first five entries

Having these four data elements it is possible to layout the network's infra-structure in a decentralized form, it is however useful to aggregate a traffic light controller to each intersection. This is discussed in the next section.

3.4 TRAFFIC LIGHTS

This work opts for a FSM (finite-state-machine) based implementation to carry out the various controllers mentioned throughout this paper. The finite-state-machines coding style used in this work are based in that of *Sklyarov, Valery*. http://ieeta.pt/~skl/SDR_V University of Aveiro, department of Electronics, Telecommunications and Informatics [17]. Alternatives include the use of soft-core processors such as in that in the picoblaze microcontroller [20].

Traffic light controllers can vary in behavior according to timing specifications, light sequencing and sensing mechanisms. This section starts off by discussing the basic elements which comprise a simple traffic light model. After identifying these, we pass on to a sensor based traffic light controller extension and its corresponding implementation.

3.4.1 TRAFFIC LIGHT MODEL ELEMENTS AND DATA SET

In order to provide design flexibility, a traffic light controller for each route is considered therefore a single light traffic controller is composed of the following elements:

- On/Off control.
- Light status (Red, Yellow and Green) and associated timeout values.
- Timer.

By assigning a controller to each route and by coordinating these, an intersection level traffic light controller is easily attainable as will be shown briefly. Consequence of this approach is that all

previously identified elements have to be replicated for each route (or traffic light). Although these elements seem rather self-explanatory, it will be seen further on that a non-zero initialization timer values may need to be calculated according to the implemented traffic light coordination scheme.

VHDL Description

In order to utilize FPGA resources effectively, elements that are needed to be accessed by others than the traffic light controller itself (e.g. by incoming vehicles), as are the on/off and light statuses were implemented in distributed resources therefore being easily accessible to multiple circuits simultaneously. On the other hand the timer for each intersection, as well as the corresponding timeout values were stored in Block RAM, whose access is limited in number due to a limited port access (single or double port) and due to read/write access latency. The corresponding VHDL data structure to store the light and on/off status content for each intersection is the described as followed (VHDL Description 3.4.1-1):

```

1  constant MAX_INTERSECTIONS    : natural := 50; -- Total limit of intersections
2  constant TIMER_MAX            : natural := 32; -- In seconds
3
4  type LIGHTS_ENUM is (R, Y, G); -- Red, Yellow and Green.
5
6  type ROUTE_LIGHTS_TYPE is array(ROUTES_ENUM'left to ROUTES_ENUM'right) of
   LIGHTS_ENUM;
7
8  type ROUTE_LIGHTS_ON_TYPE is array(ROUTES_ENUM'left to ROUTES_ENUM'right)
   of STD_LOGIC;
9
10 type ROUTE_LIGHTS_REC_TYPE  is RECORD
11  lightstatus    : ROUTE_LIGHTS_TYPE;
12  onoff         : ROUTE_LIGHTS_ON_TYPE;
13 end RECORD;
14
15 type INTER_ARRAY_SIGNA_TYPE is array(1 to MAX_INTERSECTIONS) of
   ROUTE_LIGHTS_REC_TYPE;
```

VHDL Description 3.4.1-1 Intersection traffic light controller VHDL data structure

The previously defined UTS infrastructure (see figure 3.2.2-6) has a total of fifty intersections therefore in limit we should have this number of intersection level traffic light controllers. The number of traffic light controllers can be easily edited by defining a new UTS network infrastructure and editing the MAX_INTERSECTIONS constant accordingly. The memory components depths have to be manually edited.

In this case the random-access memory element(Figure 3.4.1-1) that stores a timer value for each intersection is 5-bit wide by 200 deep capable of storing up to four counters per intersection (fifty intersections and one counter for each possible route) and counting up to thirty-two (seconds) for each timer. The random-access memory element (Figure 3.4.1-2) that stores the timeout values of each traffic light of the entire UTS network is 15-bit wide by 200 deep. The 15-bit memory word is divided into three 5-bit values that represent the timeout values for the Red, Yellow and Green color status respectively from the MSB to the LSB. It can therefore be observed that the stipulated maximum period of time of each light status is 32 seconds.

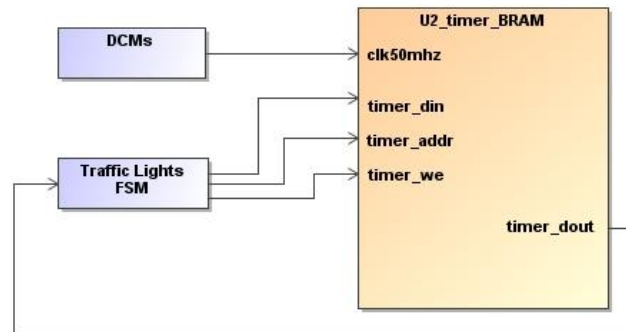


Figure 3.4.1-1 Timer values for each traffic light memory component

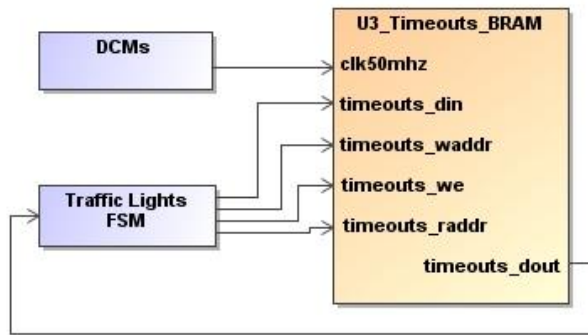


Figure 3.4.1-2 Timeout values for each traffic light memory component

As a practical example, describing the first five intersections can therefore take the following form (VHDL Description 3.4.1-2) in which the first intersection has no traffic lights active, intersection number two has only the north traffic light active and as another example intersection four has the east, north and south traffic lights active.

```

1 signal inter_s_signal : INTER_ARRAY_SIGNAL_TYPE :=(
2   -- ((LIGHTS),(LIGHTS ON/OFF  ))
3   ((R,G,R,G),('0','0','0','0')),--1
4   ((R,G,R,G),('0','1','0','0')),--2
5   ((R,G,R,G),('0','1','0','0')),--3
6   ((R,G,R,G),('1','1','1','0')),--4
7   ((R,G,R,G),('0','1','0','0')),--5
8   ...);

```

VHDL Description 3.4.1-2 Intersection traffic control description

The corresponding initialization contents of the timer and timeout values memory for the first intersection are shown in figure 3.4.1-3 and 3.4.1-4, respectively. These will be explained in depth next.

```

; 5-bit wide by 200 deep ROM
memory_initialization_radix = 2;
memory_initialization_vector =
;3210 1
00001,
00000,
00001,
00000,

```

Figure 3.4.1-3 Timer RAM initialization of first intersection

```

; 15-bit wide by 200 deep ROM
memory_initialization_radix = 2;
memory_initialization_vector =
;43215432154321 1
0100000100000001,
0100000100000001,
0100000100000001,
0100000100000001,
0100000100000001,
...

```

Figure 3.4.1-4 Timeout RAM initialization of first intersection

By examining figure 3.4.1-3 it can be observed that the corresponding East and West route timer values are non-zero values. The meaning of this is given in the following section where the implementation of a given traffic light controller scheme for a basic traffic light controller is discussed.

3.4.2 BASIC TRAFFIC LIGHT MODEL

Considering a four-way traffic intersection, two practical traffic light sequencing schemes are analyzed. In the first arrangement (whose light cycle is illustrated figure 3.4.2-1) opposing (N-S and E-W) traffic controllers are replicates in terms of behavior. In a second arrangement each traffic light has an independent cycle as observed in figure 3.4.2-2. These figures are divided into time-frames which illustrate how light state changes evolve in function of time.

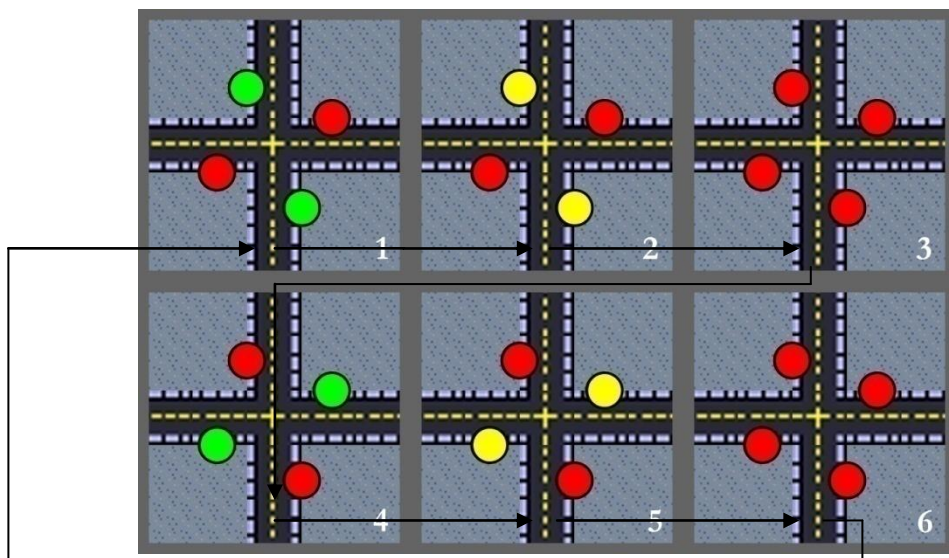


Figure 3.4.2-1 Cross-linked traffic light scheme

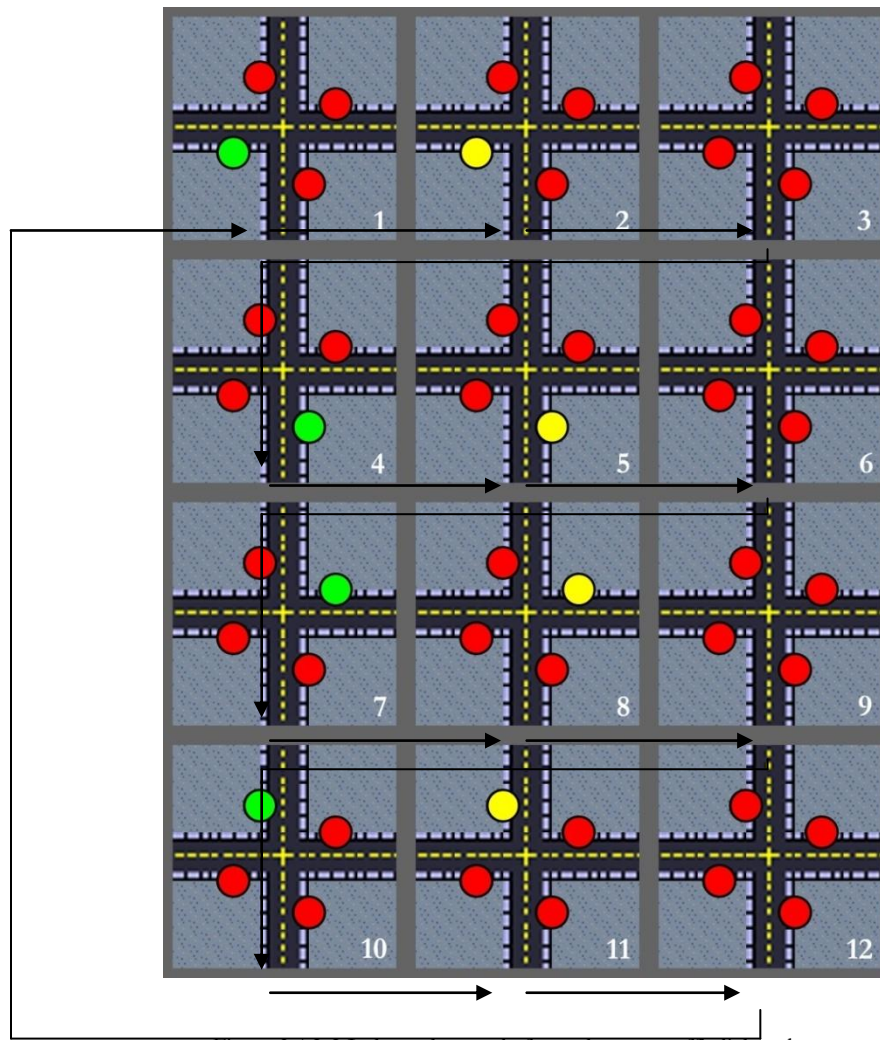


Figure 3.4.2-2 Independent cycle for each route traffic light scheme

In order to implement the first traffic light controller arrangement, we can start by considering the initial light states to be (R, G, R, G) as figure 3.4.2-1 suggests. These light states correspond to the East, North, West and North routes respectively (this association is always considered unless stated otherwise). Having chosen an initial state, the next steps considered are to obtain valid timeout and timer offset values for each route/traffic light. For simplicity sake we can assume that the timeout values for each light state are the same independently of the route. As an example it can be stipulated that the green light state's timeout value is five seconds and the yellow light state's timeout value is two seconds. By observing figure 3.4.2-1 it is obvious that the red state's timeout value must be superior to the sum of the green and yellow states timeout values since the red state

remains active in the course of timeframes. It can also be observed that since there are two timeframes (timeframe three and six in figure 3.4.2-1) in which all traffic lights are red (safety measure) this necessarily implies using an offset value. This is easily perceived by noting that in timeframe one the East and West route traffic lights have non-zero timer values since they were already in the red state in the previous time-frame, which is time-frame six in the figure. Assuming a duration of one second for timeframes three and six, this implies that in timeframe one the North and South traffic lights are in the red state with a one second offset, therefore their timers mustn't start at zero but with an offset value of one. Since the red lights must persist further during an additional green, yellow and "safety red" cycles it can be concluded that the timeout for the red state is $1 + T_{\text{green}} + T_{\text{yellow}} + 1 = 9\text{sec}$. The result for the first sequencing scheme is hereby:

	<i>East</i>	<i>North</i>	<i>West</i>	<i>South</i>
Initial state	R	G	R	G
Timer (offset) (sec)	0	1	0	1
Red timeout (sec)	9	9	9	9
Yellow timeout (sec)	2	2	2	2
Green timeout (sec)	5	5	5	5

Table 3-1 Traffic light controller parameters for the first traffic light scheme

Regarding the second arrangement we can assume, as the previous case, a timeout value for the green and yellow states to be five and two seconds respectively. Considering the initial state to be (G, R, R, R) correspondent to timeframe one in figure 3.4.2-2, the timer for every route except for the East route traffic lights involve offset values. As in the first arrangement this can be easily perceived since in the previous state (timeframe 12 in figure 3.4.2-2) these traffic lights were in the same state. Observing figure 3.4.2-2 the offset value for the other traffic light can be obtained by back-tracking the traffic light states in previous timeframes summing their durations. The resulting values are presented in the following table (Table 3.4.2-2):

	<i>East</i>	<i>North</i>	<i>West</i>	<i>South</i>
Initial state	G	R	R	R
Timer (offset) (sec)	0	1	9	17
Red timeout (sec)	25	25	25	25
Yellow timeout (sec)	2	2	2	2
Green timeout (sec)	5	5	5	5

Table 3-2 Traffic light controller parameters for the second traffic light arrangement

By analyzing both arrangements it can be stated that by assigning different timeout values for each route, complex timing schemes can be implemented offering a high level of design flexibility.

Since Block RAM memory components are used to store the timer and timeout values this makes it impossible to realize a truly parallel description of every traffic controller in VHDL, i.e. it is impossible to have a controller functioning independently and in parallel for each intersection, the reasons as to why this happens will be discussed next.

As stated before using distributed memory, instead of Block RAM would lead very quickly to a resource over-usage for the current FPGA (Spartan 3E-500). The consequence of using Block RAM based memory components means that only one FSM can address a specific memory component at a time. Since we can have a large number of traffic light controllers (for the present case two hundred, considering we have four in each intersection and a total of fifty intersections), this can present itself as a bottleneck in terms of processing speed. A possible alternative is implementing a pipelined architecture for the traffic light controller to speed up the processing speed, or using multiple or dual port memory components (which is more feasible), but these fall out of the scope of this work, since performance isn't being focused on. Instead, this paper recurs to a tile-sliced controller being the only obvious drawback the limited processing speed since this FSM has to process every traffic light controller parsing the entire network. Having this in mind a state chart of the finite state machine used to implement the UTS's traffic light controllers is illustrated in figure 3.4.2-4. An in depth analysis of this FSM an each one of its states is given next.

As each second elapses the FSM is triggered to process every traffic light controller in the UTS network, increment its timer and change the light's state incase a timeout occurs. Since the

previously discussed time-slicing mechanism is used, it is useful to assess the FSM's capability to process every traffic light between second "ticks" as this is fundamental for the traffic light controller's correct operation. Taking in consideration that the state machine at hand is fairly simple a direct analysis without any formal validation can be attempted. In order to achieve an analysis it can be observed that the longest processing sequence for a single traffic light occurs when a timeout occurs. This corresponds to the $S_0:S_1:S_2:S_3:S_4:S_6:S_7:S_8\&S_9$ state transition sequence when there are unprocessed routes in the present intersection and $S_0:S_1:S_2:S_3:S_4:S_6:S_7:S_8\&S_9:S_{10}$ when all routes in the present intersection have been processed. As an example, let's consider a hypothetical scenario in which we have an UTS network with one thousand intersections in which all have four ways and every traffic controller has reached a timeout. Considering the longest processing sequences above it can be seen that the first sequence is executed three times and the fourth one time in order to process each intersection. This corresponds approximately to twenty-nine state transitions to process each intersection. Since we consider one thousand intersections this amounts to gross estimate of twenty nine thousand state transitions necessary to process the entire network. Further assuming a state machine clock of 12.5 MHz and that each state transition takes a single clock period then the total processing time for the UTS is approximately 2.32 mS. Since this represents the maximum delay between intersection updates, it can be considered acceptable when taking into account real life hardware communication or processing latencies and factors for a traffic-light and traffic network user interaction (e.g. a driver's average reaction time is 2s). This delay may however present itself as an obstacle when taking into account communications between traffic management controllers in a real life situation, in that case investment in fully parallel implantation should be made.

Another useful analysis is estimating the maximum number of traffic lights the controller can process in a one second interval, since this is the maximum processing time to assure a correct operation of the controller. Conserving the conditions mentioned above, the maximum number of intersections that can be processed in one second is approximately 431000 ($12.5E6/29$) traffic lights.

In regard to the previously stipulated FSM functioning clock frequency it must be noted that 12.5Mhz is rather conservative since the Nexys2 board offers a standard clock signal of 50Mhz, which can be manipulated through DCMs (Digital Clock Manager) to achieve higher frequencies. The downside of operating at higher frequencies is that more logic resources are needed in order to reduce electrical signal propagation related latencies. Hereby we can identify a tradeoff between performance and resource usage.

Following figure 3.4.2-4, a description of each state is given next in table 3.4.2-3:

State	State Description	Previous States	Next States
S0	The FSM remains in this state while the seconds tick remains low. Once the seconds tick goes high the machine changes state to S1.	Reset; S10	S1
S1	If there is no traffic light in the current route (off) then the next state is S8, otherwise the FSM advances to S2.	S0; S8&S9; S10	S2; S8&S9
S2	Since the timer and timeout memory components have a one clock cycle read access latency these are addressed before-hand. The state machine advances to S3.	S1	S3
S3	The FSM checks the current light status and reads the corresponding timeout value from the timeout memory component and advances to S4.	S2	S4
S4	By reading the current timer value from the timer RAM and comparing it to the previously read timeout value it can be observed if a timeout occurred. If so the next state is S6 else it is S5.	S3	S5;S6
S5	The FSM increments the timer value and writes this value to the timer RAM by asserting the corresponding write enable signal. The next state is S8.	S4	S8&S9
S6	The FSM resets the timer to zero, writing this value to the timer RAM by asserting the corresponding write enable signal. The next state is S7.	S4	S7
S7	The traffic light changes to the next color state following the Green, Yellow and Red color sequence. The next state is S8.	S6	S8&S9

S8&S9	The FSM checks if all routes/traffic light controllers in the current intersection have been processed, case affirmative it reset to the first route (East) and transitions to S10, otherwise it switches to the next route and transitions to S1. These operations have been separated in two states (S8&S9) since too many operations in a single state require an extra effort to synthesize due to timing constraints.	S1; S5; S7	S10
S10	S10. If the FSM has processed every intersection it moves on to S0, otherwise it increments the intersection and processes it transitioning to S1.	S8&S9	S0; S1

Table 3-3 – Traffic light FSM state description

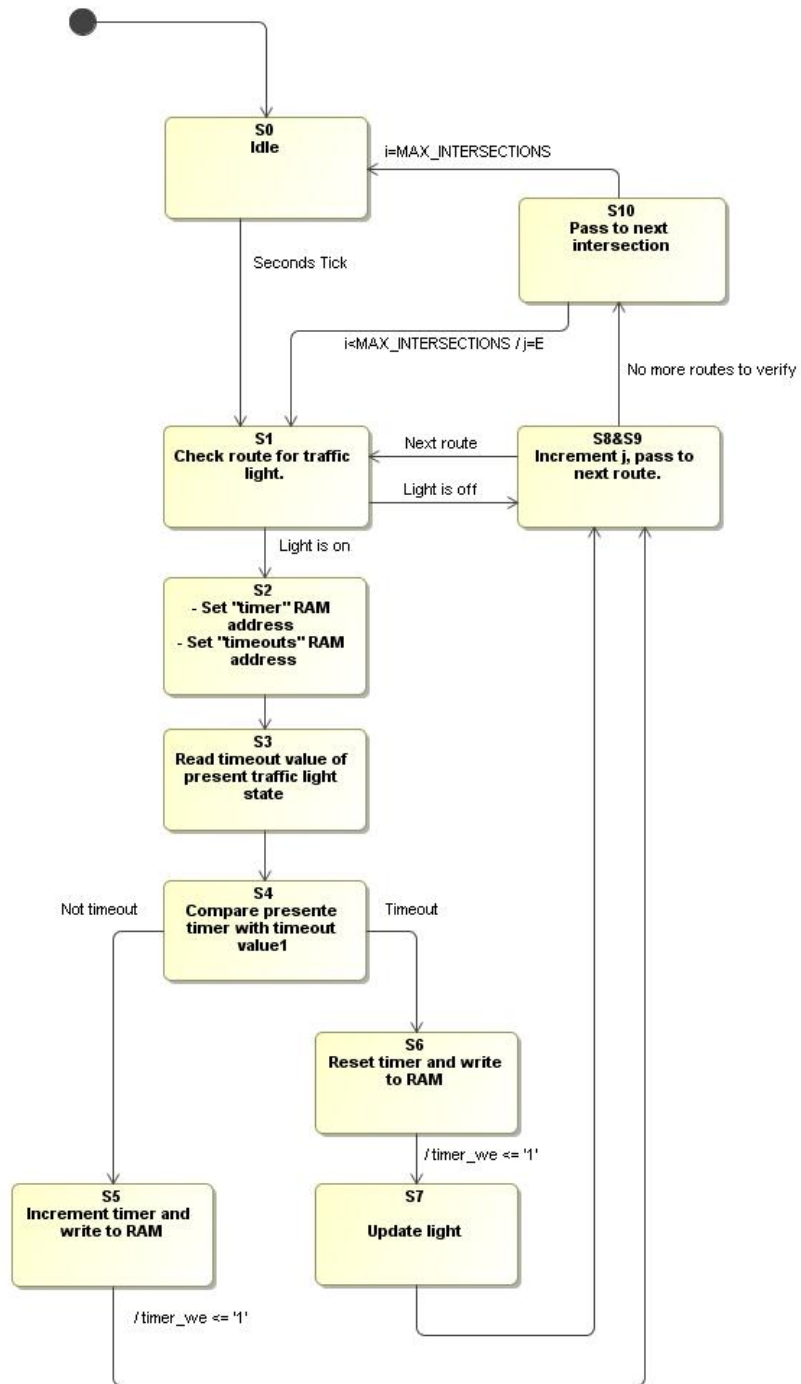


Figure 3.4.2-3 Traffic light control state machine diagram

3.4.3 INTELLIGENT TRAFFIC LIGHT MODEL

In order to assess vehicle-traffic light controller interaction an “intelligent” traffic light controller is designed in this section based on vehicle sensing similar to that of induction loops.

The previously defined traffic light control algorithm has a potential flaw in a practical efficiency point of view. In the scenario that a route has no waiting or incoming vehicles there is no sense in activating a green light for that route. This section addresses this issue by extending the control data for each intersection with an incoming signal bit for each route that has the value ‘1’ whenever there is an incoming vehicle at a predefined distance from the intersection’s center and ‘0’ otherwise. This work assumes that this signal is asserted by the vehicle controller, but perhaps a more valid case would be to implement an independent controller which actually monitored each vehicle with the objective of detecting its approximation, therefore reducing the complexity of the vehicle controller and maintaining its functionality. The resulting state machine diagram is illustrated in figure 3.4.3-1 and includes both sensing and non-sensing traffic light controller models, i.e. it represents a superposition of the previously defined controller. In regard to the sensing operation when sensing is activated the traffic light controller doesn’t take further action (i.e. doesn’t increment route timers and doesn’t process timeouts) until there is indication that a vehicle is incoming or waiting on a red signal. When sensing is de-activated the behavior is similar to that of the previous traffic light controller.

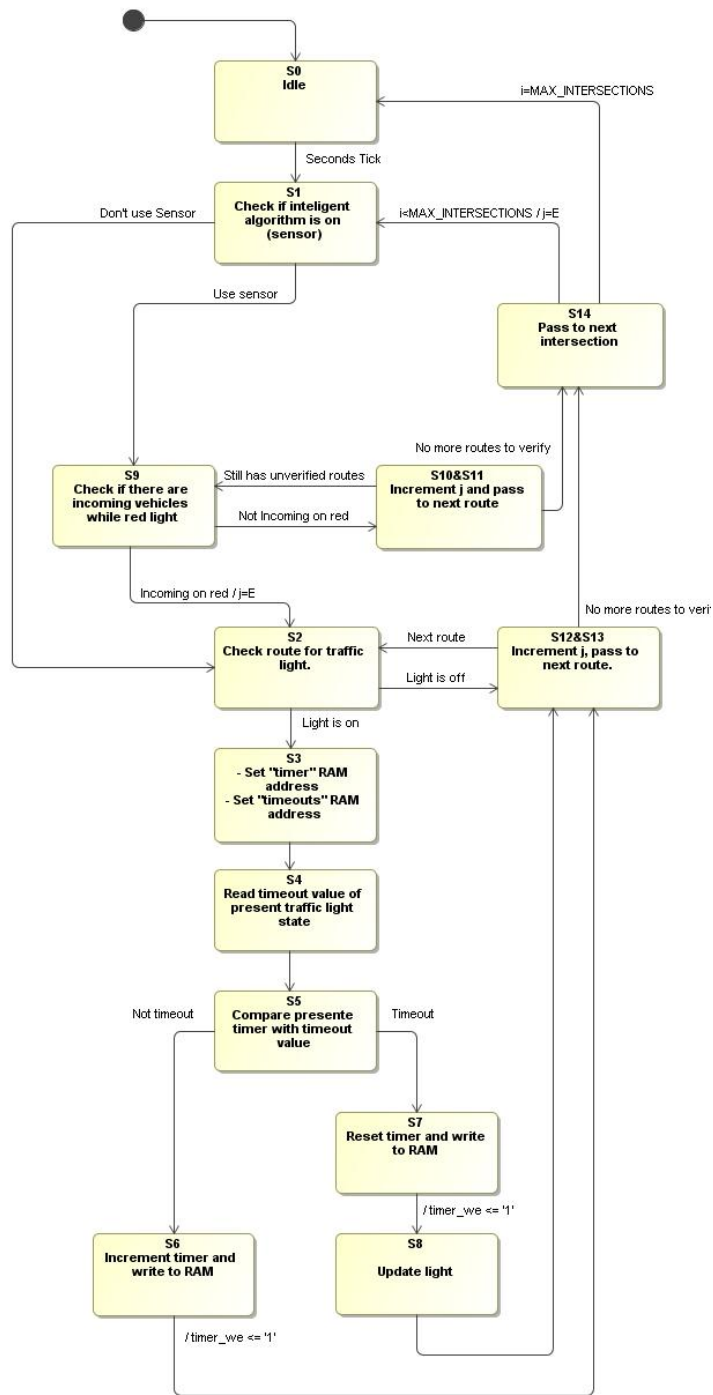


Figure 3.4.3-1 - Intelligent traffic light state machine diagram

As can be seen by observing the state chart and comparing it to that of figure 3.4.2-4, the only difference is the addition of the S1, S9 and S10&S11 states in relation to the previous traffic light controller state chart. A description of each new state follows:

State	State Description	Previous States	Next States
S1	Checks if sensing is activated for the current intersection transitioning to S9 case affirmative otherwise it transitions to S2 from where the controller takes a similar action to the previous traffic light controller.	S0; S14	S2; S9
S9	Monitors the current route and polls the correspondent incoming signal. If this signal is high it processes the traffic light controller which starts by transitioning to S2, otherwise it transitions to S10 being the equivalent of taking no action, holding the current timer and state values.	S1; S10&S11	S2; S10&S11
S10&S11	Similar operation as the previous controller's S8&S9 states.	S9	S9; S14

3.5 VEHICLES

This chapter starts by discussing the basic elements which allow a vehicle model to be implemented according to the UTS infrastructure and the traffic light implementations discussed in the last sections. In section 3.5.2 we discuss a random route taking vehicle model, which is useful for an initial evaluation of vehicle mobility in a general UTCS. A user controllable vehicle is discussed in sec. 3.5.3, this model has the ability to calculate and take the shortest path between any two intersections of the UTS network.

3.5.1 VEHICLE MODEL ELEMENTS AND DATA SET

A basic vehicle model needs to be created in order to assess vehicle navigation in the previously described UTS network. Taking into account the UTS network infrastructure and the traffic light controllers discussed in sections 3.3 and 3.4 the following elements were identified in order to implement a viable vehicle model:

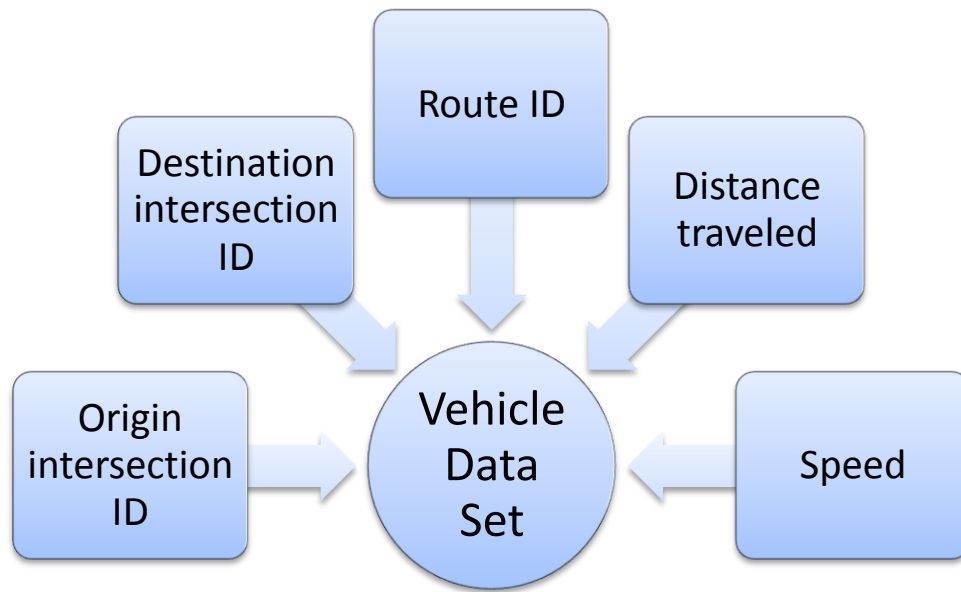


Figure 3.5.1-1 Basic vehicle model data set elements

The origin and destination intersection identification elements follow the intersection numbering previously mentioned. The route identification can take E, N, W and S values which were initially referenced and described in chapter 3.3. The vehicle's traveled distance can be measured by taking into consideration that each tile map that divides the network's infrastructure contains 16 pixels, so as to simplify we consider the vehicle's traveled distance to be measured in pixels. The present work considers vehicle movement in a discrete fashion. Being so each vehicle updates its position and increments its distance according to its speed parameter and a clock division circuit (figure 3.5.1-2). Consequently vehicles with a greater speed update their status/position with a greater frequency and vehicles with lesser speed with a lower frequency. In this case each vehicle increments its position and distance one pixel. An alternative is every vehicle updating its status/position at the same frequency yet having vehicles with a greater speed incrementing their position and distance by a larger value. The advantage of the first implementation scheme (the current work's implementation) is a smoother graphical animation and the advantage of the alternative case is a less complex controller.

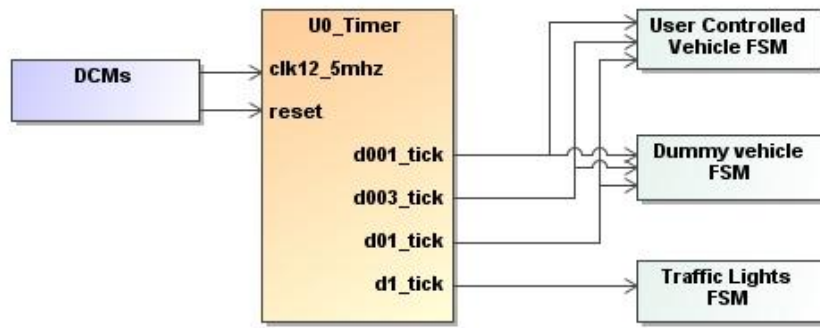


Figure 3.5.1-2 Clock divider/tick generation circuit

Being important due to the graphical simulation, the vehicle's position coordinates can be obtained by reading the origin intersection's coordinates and summing/subtracting the vehicle's distance according to its current route, therefore the vehicle position coordinates are not considered in the previously identified model elements. Keeping a record of these would avoid calculations however considering a large number of vehicles the resource occupation involved would overwhelm the resource scarcity of the current FPGA (Spartan 3E-500).

VHDL description

In order to implement the previous description the following VHDL data structure was defined (VHDL Description 3.5.1-1), as in the intersections and traffic lights here vehicles data sets are aggregated in an array:

```

1  constant MAX_VEH_DISTANCE : natural := 9; -- 512
2  constant MAX_VEHICLES      : natural := 90; -- Stipulated maximum number of vehicles
3  type ROUTES_ENUM is (E, N, W, S);
4
5  type VEH_REC is RECORD
6  distance   : UNSIGNED(MAX_VEH_MILEAGE-1 downto 0);
7  origin     : natural range 1 to MAX_INTERSECTIONS;
8  destination : natural range 1 to MAX_INTERSECTIONS;
9  route      : ROUTES_ENUM;
10 speed      : natural range 0 to 2;
11 end RECORD;
12

```

```
13 type VEH_ARRAY_TYPE is array(1 to MAX_VEHICLES) of VEH_REC;
```

VHDL Description 3.5.1-1 Vehicle data set

The clock division circuit (figure 3.5.1-2) generates time ticks (asserted during one clock cycle) with 0.01s, 0.03s, 0.1s and 1s periods. These are organized in a 3-bit array signal in which the LSB corresponds to the 0.01s tick and the MSB to the 0.1s tick signals. Therefore the vehicle's speed signal can be used as an array index so that a speed parameter of "0" is the fastest update rate (0.01 sec) and a speed parameter of "2" the slowest (0.1 sec).

Having identified and implemented a description which holds the necessary vehicle data set elements it is now necessary to design a controller which must use such a data set. This is done in the following sections.

3.5.2 DUMMY VEHICLE MODEL

We start with a basic vehicle model in order to assess the interoperability between the UTS network infrastructure and traffic light control elements (studied in sections 3.3 and 3.4) as well as the vehicle data set defined in the last sub-section. Having this in mind this sub-section describes a dummy (random route taking) vehicle model which has the following features:

- Various vehicle speed classes.
- Random route taking decision.
- Vehicle-following rules.
- Respect the right-hand rule at intersections.

The resulting controller's model can be observed by means of a state chart (figure 3.5.2-1). As in the traffic light controller's case this finite-state-machine is implemented in a time-sliced scheme for the same reasons discussed earlier. Since this FSM is rather complex relative to the traffic light's

controller case, it is useful to give an architectural overview of its state chart. States S4, S14, S15 and S16 process the vehicle when it has reached the end of its current route and therefore has to decide which route to take upon at its destination intersection. S5 defines the distance at which the “induction loops” mentioned in section 3.4.3 are situated relative to the intersections center (see figure 3.5.2-2) and also serves as a safety distance check from which each vehicle is effectively conditioned by the intersection’s traffic lights or the right-hand priority rule. States S7, S8 and S9 implement the traffic light respecting and right-hand priority rule. States S10, S11, S12 and S13 implement vehicle following rules, in which a faster traveling vehicle is forced to maintain a minimum safety distance between itself and an eventual vehicle traveling ahead of it. The chosen method to implement vehicle-following is to scan every other vehicle checking if it is found to be in the same intersection and route as the current vehicle being processed. Case affirmative both vehicle’s distances are subtracted and the result is compared to a predefined minimum safety distance between vehicles. If this distance is compromised the current vehicle shouldn’t advance. The state group formed by S17, S18, S19, S20 and S21 update the vehicle’s position coordinates and increments it’s traveled distance.

An in-depth description of each state follows next:

State	State Description	Previous States	Next States
S0	The state-machine stays in S0 until the 0.01 second tick signal is high. Since all vehicle update ticks are synchronous the 0.01 second tick covers every vehicle. When the 0.01s tick is high the controller stores a copy of the clock division circuit’s output in order to assess other vehicle update periods besides 0.01s and advances to S1.	Reset; S22	S1
S1	The controller checks, according to the current vehicle’s speed parameter if the corresponding update tick is asserted and in the affirmative case it proceeds to S2 otherwise the controller processes the next vehicle by transitioning to S22.	S0; S22	S2; S22

S2	Since the vehicle CAM and the intersection position coordinates memory components have read/write access latencies the controller addresses these at S2, this also avoids code replication in the various following states. The state machine transitions to S3.	S1	S3
S3	In this state the FSM evaluates whether or not the vehicle has reached the end of the current route by comparing the current vehicle's traveled distance with the current route's length. If the vehicle has reached the end of the route the state machine transitions to S4 otherwise it transitions to S5.	S2	S4; S5
S4	Since the vehicle has reached the end of the current route, it has to decide its destination route in order to assert the vehicle's position as well as decide on what route to take. Due to memory access latency the destination intersection position coordinates ROM is addressed in this state. The FSM advances to S14.	S3	S14
S5	In the present work a safety distance in relation to the center of each intersection is considered (see figure 2.5.2-2 where a safety distance of 16 is illustrated). This is useful so that each vehicle can flag the incoming vehicle signal discussed in section 3.4.3, in this perspective the safety distance represents the distance from the intersection at which the induction loop is placed. When a vehicle enters this safety area it also respects the traffic lights or the right-hand priority rule, this behavior pretends to mimic the behavior real drivers have since attention is only given when arriving at an intersection. In case the vehicle hasn't entered the safety area the FSM transitions to S10,	S3	S6; S10

	otherwise it transitions to S6.		
S6	The vehicle is inside the intersection's safety area. In this state the vehicle flags the incoming vehicle in the corresponding route of its destination intersection. The FSM then transitions to S7.	S5	S7
S7	The vehicle is inside the intersection's safety area. It checks the existence of a traffic light, if existent the FSM transitions to S9, otherwise it advanced to S8 identifying the route situated to the current route's right-hand side by means of a look-up table which holds the correspondent right-hand side route according to the current vehicle's route (VHDL Description 3.5.2-1).	S6	S8; S9
S8	Since there is no traffic light, the vehicle should check if there is any incoming vehicle in the route situated to the current route's right-hand side, which was obtained in S7. Therefore if the incoming signal in the corresponding route is set, the vehicle should not advance and the FSM transitions to S22. If the incoming signal bit is cleared the vehicle can advance and the FSM advances to S10.	S7	S10; S22
S9	In this state the vehicle is conditioned by the corresponding destination intersections traffic light status. If the light is green the vehicle should advance and the FSM advances to S10 otherwise it transitions to S22.	S7	S10, S22
S10	This state along with S11, S12 and S13 form what is considered as vehicle-following mentioned earlier. In this state the FSM checks if the	S5; S8; S9	S11; S13

	currently vehicle being checked is in the same state as the currently processed vehicle. If affirmative the FSM advances to S11 otherwise it transits to S13.		
S11	The FSM checks if the currently vehicle being verified for vehicle-following is in the same route as the currently processed vehicle. If affirmative the FSM advances to S12 otherwise it transits to s13.	S10	S12; S13
S12	The current vehicle being verified for vehicle-following is in the same route as the currently processed vehicle therefore in this state their distances are subtracted and the result is compared to a minimum safety distance value. If this distance is comprised the current vehicle being processed shouldn't advance and therefore the FSM transits to S22, otherwise the vehicle may advance and goes to S13.	S11	S13; S22
S13	This state increments the vehicle being verified for vehicle-following to the next vehicle. If all vehicles have been verified it advances to S17 otherwise it transits to S10.	S10; S11; S12	S10; S17
S14	In this state the current vehicle being processed chooses a random route. This is a pseudo-random decision based in a LFSR (Linear Feedback Shift Register) circuit. More information about this implementation can be found in http://www.geocities.com/SiliconValley/Screen/2257/vhdl/lfsr/lfsr.html [21]. A two bit random output is extracted from the circuit which then indexes four word look-up table which holds different route values. S14 compares the random route output, checks if the route exists in the target intersection and assures that it doesn't correspond to the	S4	S14; S15

	vehicle's current route, which would result in a "u-turn". The FSM self-transits to S14 until a valid route has been acquired, once this happens it advances to S15.		
S15	In this state the current vehicle being processed validated the intersection transfer by updating the vehicle's data set such as the vehicle's origin intersection, destination intersection and route. The vehicle incoming signal (discussed in section 3.4.3) is also cleared. The FSM advances to S16.	S14	S16
S16	Since vehicles with a higher speed are updated at a higher frequency than those of lower speeds, a problem arises when a higher speed vehicle follows a lower speed vehicle into an intersection and transits to the same route. Since originally each vehicle sets it's traveled distance to zero, this means that the vehicle-following model comprised of states S10, S11, S12 and S13 doesn't behave correctly, allowing the faster vehicle to overtake the slower one. Since there is no direct method of knowing which vehicle entered the route first, a rudimentary solution was found to correct this problem. This solution considers initializing each vehicle's traveled distance instead of resetting it to zero. Therefore slower vehicles start off with a small non-zero traveled distance thereby avoiding this problem. The state machine advances to S17.	S15	S17
S17	The FSM reads the intersection position coordinates BROM and calculates its current position coordinates according to the current route as to have a correct graphical presentation. The next state is S18.	S13; S16	S18

S18	S18 divides S17's operation as to simplify the performed operations. The next state is S19	S17	S19
S19	The vehicle's traveled distance is incremented. The FSM advances to S20.	S18	S20
S20	The FSM sums or subtracts the vehicle's traveled distance to the vehicle's (or more appropriately the intersection's) position coordinates as to obtain the current position coordinates. The FSM advances to state S21.	S19	S21
S21	The FSM writes the vehicle's position to the CAM (discussed further on in section 3.7) by asserting the correspondent write enable signal. The state machine transitions to S22.	S20	S22
S22	The FSM multiplexes to the next vehicle and goes into state S1. If all vehicles have been processed it advance it advances to S0.	S1; S8; S9; S12	S0; S1

Due to the complexity of the current FSM and in the absence of a formal analysis tool the performance evaluation of the current FSM is left out. It can be said however since the current vehicle model is design to evaluate network navigation a formal evaluation of this controller isn't considered to be essential.

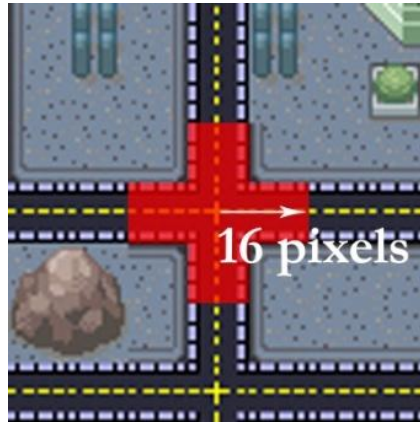


Figure 3.5.2-2 Illustration of intersection safety distance, in this case 16 pixels

```

1 type ROUTE_ROM_TYPE is array(ROUTES_ENUM'left to ROUTES_ENUM'right) of
  ROUTES_ENUM;
2 constant route_right_rom : ROUTE_ROM_TYPE := (S, E, N, W);

```

VHDL Description 3.5.2-1 ROM used to identify right-hand side route for incoming vehicle based on the vehicle's current route

3.5.3 USER CONTROLLED VEHICLE MODEL

In order to assess a real-time processing and user interaction with the UTS simulation, this work considers a user controlled vehicle. In the practical sense the development of a real time user controlled vehicle model may avoid coding complex vehicle models in the future, allowing the user to control the vehicle's behavior, this work also implements a shortest path finding calculation in real time, which can be considered in close relation to intelligent vehicles. We therefore take into account a user controlled vehicle with two basic features which are user interaction by means of a PS/2 mouse and shortest path calculation and taking between any two intersections in real time.

The operation is based on the user clicking on any intersection and the vehicle calculates and follows the shortest path between its present position (*source*) and the clicked intersection (*target*). There are many algorithms by which one can calculate the shortest path between two nodes, this paper opts for the Dijkstra's algorithm of which a full understanding can be obtained by consulting *Dijkstra's Algorithm*, From Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm [22]. For this we have to perceive the network infrastructure as a bi-directional graph in which routes are considered as edges and intersections as nodes. Shortest Path Following (SPF) is also useful taking into consideration “intelligent” vehicles that somehow have access to the traffic network's route distances, a practical application of these algorithms are the popular GPS gadgets.

In order to implement such a vehicle model, three basic “processes” were identified. One is obviously the FSM responsible for the vehicle's movement similar as the previously defined “dummy” vehicle controller. It is also necessary to perform the Dijkstra's algorithm which needs to be implemented by another FSM. Finally some sort of control is needed in order to provide user input by means of the PS/2 mouse, this is done by a third FSM. The interaction between these three processes was first modeled by means of the following activity diagram (figure 3.5.3-1).

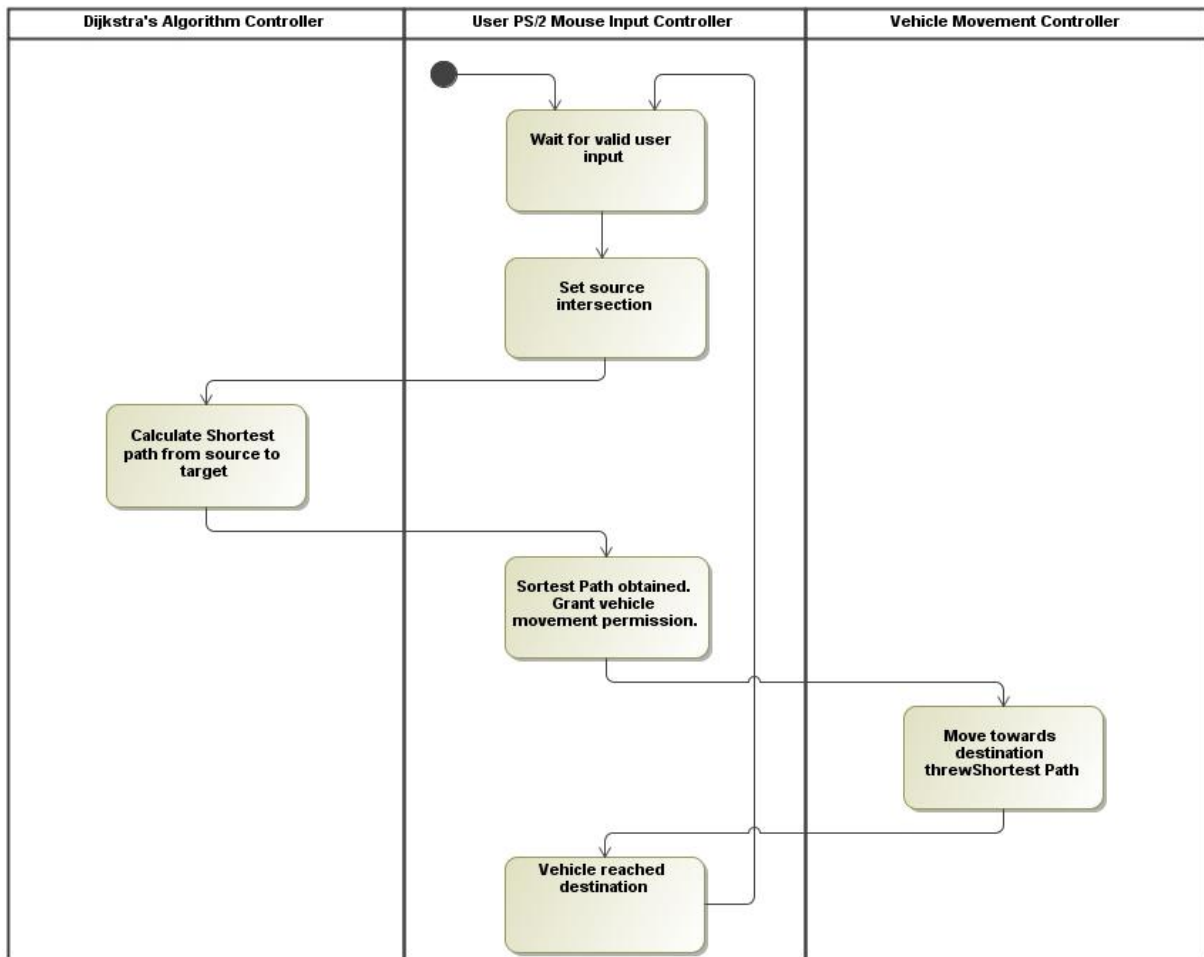


Figure 3.5.3-1 User controlled vehicle activity diagram

We will now consider these processes separately giving an in-depth description of each one and its implementation.

Vehicle Controller

The vehicle controller is in this case, a simplified version of the previous vehicle controller discussed in the last chapter, maintaining the same model elements but having a much simpler behavior. Being a simplified controller in comparison to the “dummy” vehicle controller the model considered here doesn’t incorporate any of the previously implemented priority rules such as vehicle-following, right-hand priority or traffic light rules.

VHDL Description

The previously defined vehicle model implementation utilizes single port memory components to store some of the data set elements, it doesn't make sense sharing these components since the user controlled vehicle represents a single instance and therefore there isn't any advantage taking into account the overhead and design complexity that such implementation would arise concerning memory access issues. Therefore the vehicle's position is stored in local signals i.e. in distributed memory. The VHDL description to hold these data elements for the user controlled vehicle is similar to that of previous vehicle having added a position signal as can be seen in the following description (VHDL Description 3.5.3-1):

```
1  constant MAX_VEH_MILEAGE : natural := 9; -- 256
2
3  type VEH_POS_REC is RECORD
4    x : UNSIGNED(9 downto 0);
5    y : UNSIGNED(8 downto 0);
6  end RECORD;
7
8  type VEH_REC is RECORD
9    position : VEH_POS_REC;
10   distance : UNSIGNED(MAX_VEH_MILEAGE-1 downto 0);
11   origin   : natural range 1 to MAX_INTERSECTIONS;
12   destin   : natural range 1 to MAX_INTERSECTIONS;
13   route    : ROUTES_ENUM;
14   speed     : natural range 0 to 3;
15 end RECORD;
16
17 signal uveh_state_reg, uveh_state_next : STATE_TYPE;
18 signal uveh_idle      : STD_LOGIC;
19 signal uveh           : VEH_REC := (TO_UNSIGNED(05*16, MAX_VEH_MILEAGE),
    34, 35, W, 1);
```

VHDL Description 3.5.3-1 Code description for the user controlled vehicle

Since the user controlled vehicle has an independent controller, i.e. it doesn't share the controller discussed in 3.5.2 for the "dummy" vehicle, it is unable to obtain information of the intersection position coordinates using the previously created intersection position coordinates RAM. A quick solution was to configure the intersection position coordinates memory component with a second port. Although this is a valid solution it must be stated that Block RAM doesn't support more than two ports for each memory component, therefore a future-proof solution would be to implement another better suited solution (e.g. shared bus with respective controller), but this falls out of the scope of this work and is left as a reference only.

The corresponding FSM's state chart for the user controlled vehicle is presented next (figure 3.5.3-2) followed by a detailed description of each state:

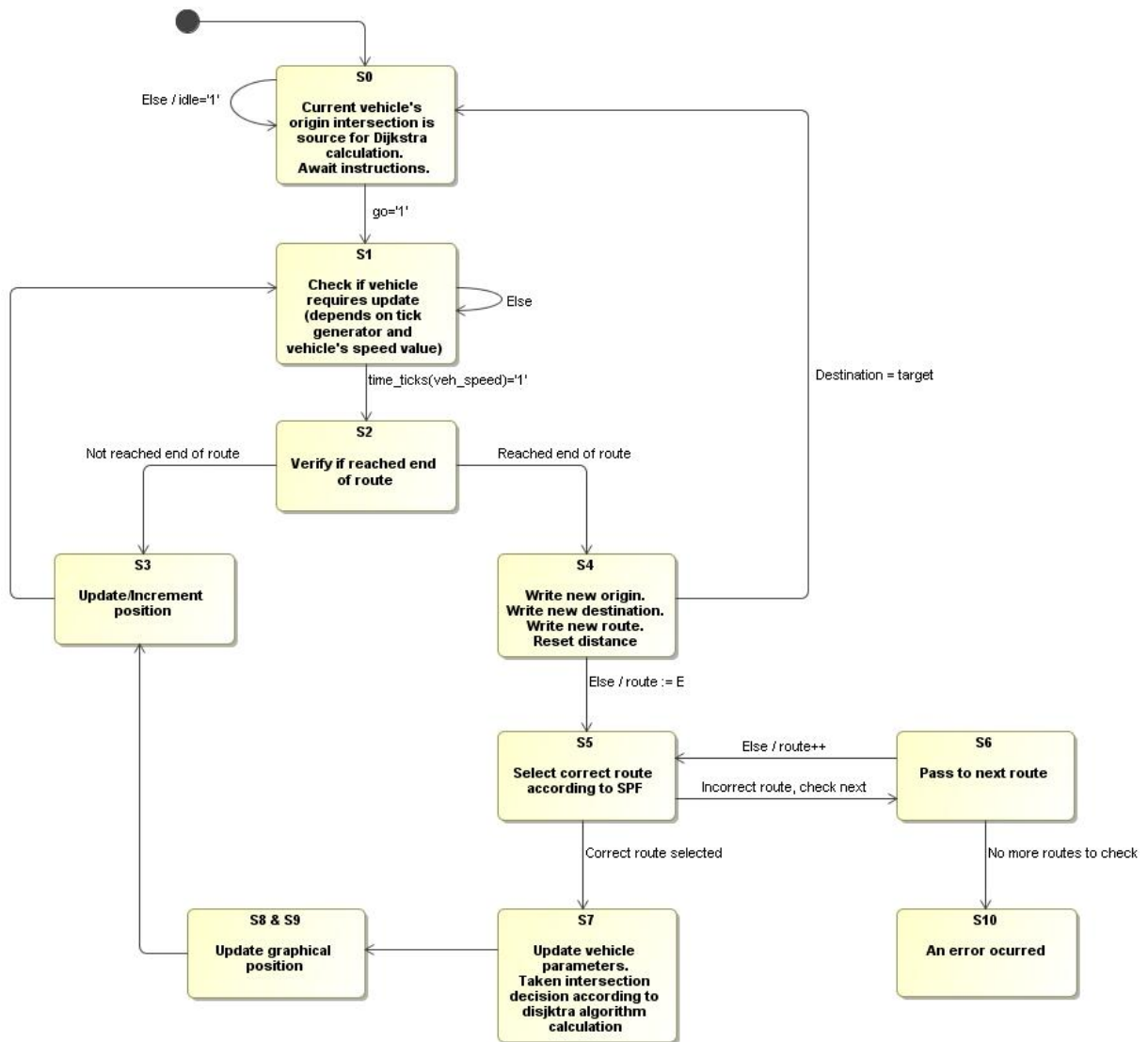


Figure 3.5.3-2 User controlled vehicle movement FSM's state chart

State	State Description	Previous State	Next State
S0	The vehicle polls the <i>veh_go</i> bit signal, which is controlled by the user input controller. When this signal is asserted this means a softest path exists and has been calculated therefore the vehicle may proceed towards the destination intersection advancing to S1.	Reset; S0; S4	S1
S1	As in the “dummy” vehicle controller, this controller updates the vehicle’s position/status at a rate which is dependent on its speed parameter. Therefore the vehicle waits until the correspondent time tick signal is set. When this bit signal is set the FSM advances to S2.	S0; S1; S3	S1; S2
S2	In this state the vehicle verifies if it has reached the end of the current route, if so it advances to S4, otherwise it advances to S3.	S1	S3; S4
S3	In this state the vehicle updates its position. The vehicle’s traveled distance is also incremented. The FSM transitions to S1.	S2; S8&S9	S1
S4	The vehicle reads its current destination intersection, if it verifies that it corresponds to the final destination it stops by transitioning to S0, otherwise it advances to S5.	S2	S0; S5
S5	Since the vehicle has to take the shortest path, it must consult the Dijkstra’s algorithm solution, which provides the route that should be taken in the current intersection. If the current route is equal to the solution’s route, the FSM advances to S7, otherwise it increments the route by advancing to S6.	S4; S6	S6; S7
S6	The FSM increments the current route signal and advances to S5. If all routes have been processed and no action has been taken then there occurred an error and the FSM transitions to S10.	S5	S5; S10
S7	The vehicle addresses the intersection position RAM (port b) and updates its parameters such as the current origin and destination intersections and route. The FSM advances to S8.	S5	S8&S9
S8&S9	In this state the vehicle updates its position coordinates and resets its traveled distance then advances to S9. These operations are divided in two states to maintain calculations simple in each state.	S7	S3
S10	This state represents an error since the vehicle can’t follow the correct path.	S6	S10

Dijkstra’s Algorithm Implementation

This paper does not attempt to give an in-depth explanation of the Dijkstra’s algorithm, this is a matter which is easily found in appropriate literature, and one possibility is consulting *Dijkstra’s*

Algorithm, From Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm [22], where the following pseudo-code of the algorithm was extracted (figure 3.5.3-3).

```

1  function Dijkstra(Graph, source):
2      for each vertex v in Graph:                // Initializations
3          dist[v] := infinity                    // Unknown distance function from source to v
4          previous[v] := undefined               // Previous node in optimal path from source
5      dist[source] := 0                          // Distance from source to source
6      Q := the set of all nodes in Graph         // All nodes in the graph are unoptimized - thus are in Q
7      while Q is not empty:                      // The main loop
8          u := vertex in Q with smallest dist[]
9          if dist[u] = infinity:
10             break                             // all remaining vertices are inaccessible
11         remove u from Q
12         for each neighbor v of u:               // where v has not yet been removed from Q.
13             alt := dist[u] + dist_between(u, v)
14             if alt < dist[v]:                  // Relax (u,v,a)
15                 dist[v] := alt
16                 previous[v] := u
17     return previous[]

```

Figure 3.5.3-3 Dijkstra's Algorithm Pseudo-Code, extracted from http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

And a possible procedure to obtain the shortest path from source to target (figure 3.5.3-4):

```

1  S := empty sequence
2  u := target
3  while previous[u] is defined:
4      insert u at the beginning of S
5      u := previous[u]

```

Figure 3.5.3-4 Procedure to extract shortest path sequence, extracted from http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

It is however unnecessary to perform the last procedure in the present case as we are only interested in obtaining a shortest path from source to target intersections. Instead it can be noted that by exchanging source and target nodes in the algorithm, the previous node array created in the algorithm's pseudo-code is the result we are interested in since it gives indication of what route (which connects the neighboring vertex indicated in the previous node array) should be followed in order to reach, in this case, the destination intersection.

By using the formerly presented pseudo-code, a VHDL routine based on a FSM was implemented, as will be discussed next.

VHDL Description

In order to hold the necessary information to perform the Disjktra's algorithm the following signals were used (VHDL Description 3.5.2-2) as well as two memory components:

```

1 -- dijkstra's shortest path
2 type Q_TYPE is array(0 to MAX_INTERSECTIONS) of STD_LOGIC;
3 signal uveh_Q      : Q_TYPE := (others => '1');
4 signal source, target : natural range 1 to MAX_INTERSECTIONS;
5 signal dij_done, dij_idle : STD_LOGIC;

```

VHDL Description 3.5.3-2 Dijkstra's Algorithm data signals

In this description the *source* and *target* signals correspond as suggested to the target and source intersections (intentionally swapped as explained above). The *uveh_Q* signal is a binary array that indicates which intersections have been processed. The *dij_done* and *dij_idle* signals establish a simple handshaking mechanism with the user input controller, this communication has been illustrated in figure 3.5.3-1. Besides these signals two memory components were also created, one which holds correct neighbor intersection for each intersection in order to follow the shortest path between this current intersection and the target intersection, this component is denominated previous node ram. The other memory component holds distance values which represent the distance each intersection has to the target intersection. These memory components are symbolized in figure 3.5.3-5 and 3.5.2-6 respectively. The state machine used to implement the algorithm is illustrated by means of a state chart in figure 3.5.3-7.

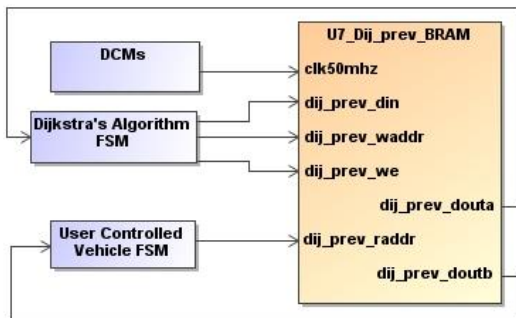


Figure 3.5.3-5 – Dijkstra previous node memory component

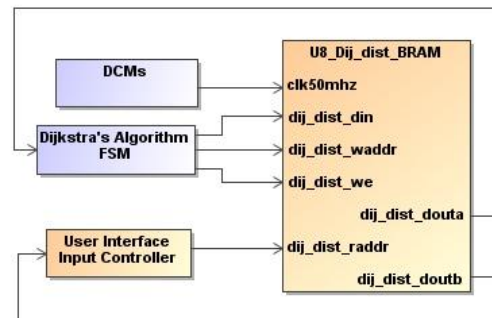


Figure 3.5.3-6 – Dijkstra distance to target value memory component

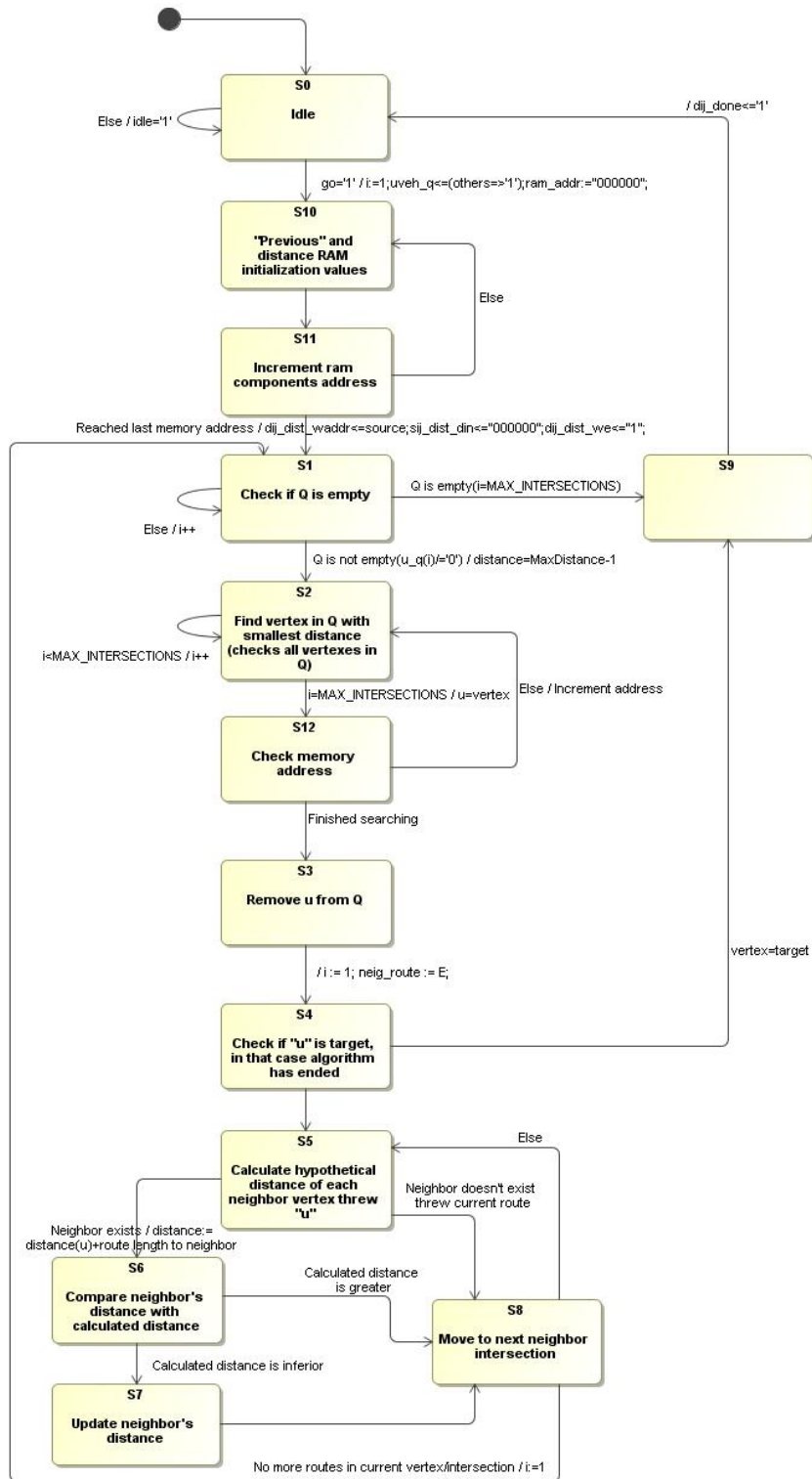


Figure 3.5.3-7 Dijkstra's Algorithm state chart

And a detailed description of each state:

State	State Description	Previous State	Next State
S0	In this state the FSM is idle, until the user input controller sets the source and target intersection ID's and brings the <i>dij_go</i> signal high, in this case the state machine transits to S1 by initializing the <i>ram_addr</i> signal used to address the previous node and distance-to-target memory components in the following states. The <i>uveh_Q</i> signal array is also initialized with '1' in every position. When the <i>dij_go</i> signal remains low, the FSM serves the user interface input controller by addressing and outputting the previous node ram element signal as will be discussed further.	Reset; S9	S0; S10
S1	This state checks if Q is empty. If any value of Q is different from '0' then Q is not empty and the FSM advances to S2 addressing the distance-to-target ram with the current element's address (this element corresponds to a non zero value in <i>uveh_Q</i>) and writing the maximum distance considered between two intersections to the distance register variable. If all elements of Q are '0' then Q is considered empty and the algorithm has calculated the shortest path distance for all intersections and the FSM advances to S9.	S8; S11	S2; S9
S2	This state in connection with S12 searches for the intersection in Q with the smallest distance. If all intersections have been processed then the intersection with the smallest distance has been obtained and the state machine advances to S3, otherwise it self-transits. This state therefore compares the currently addresses distance value with the distance register variable, if the current distance value is less than the distance variable then the distance variable takes this value as the	S1; S2; S12	S2; S12

	smallest distance-to-target index variable.		
S3	The intersection with the smallest distance-to-target is removed from Q, the FSM advances to S4.	S12	S4
S4	This state checks if the currently processed intersection (u signal) is the target, in that case the algorithm has no need in advancing and the FSM advances to S9, otherwise it continues in S5.	S3	S5; S9
S5	Calculates the hypothetical distance-to-target of each neighbor intersection of u , in case it exists. If a neighbor intersection (destination intersection for the current route) exists the FSM transitions to S6 otherwise it moves to the next neighbor/route by advancing to S8.	S4; S8	S6; S8
S6	If the previously calculated hypothetical distance-to-target is smaller than the distance the neighbor holds then the state machine advances to S7, otherwise it advances to S8.	S5	S7; S8
S7	Since the calculated distance-to-target is smaller than the current neighbor's distance, a new distance value is written to the distance ram and the current intersection is entered in the neighbor's previous intersection in the previous node ram. The FSM advances to S8.	S6	S8
S8	The FSM passes on to the next neighbor (route) and advances to S5. If all routes have been processed it advances to S1.	S5; S6; S7	S1; S5
S9	The algorithm has finished, the <i>dij_done</i> signal is set in order to inform the user controller that the calculation has completed then the FSM advances to S0 being idle.	S1; S4	S0

S10	S10 and S11 are responsible for initializing the Dijkstra's algorithm related signals as indicated in the pseudo-code. In this state the FSM writes, for the current address, the distance array with the maximum considered distance between intersections and the previous node intersections array position is initialized with a zero value, which indicate a non-existent intersection ID since these start at one. The FSM advances to S11 where it increments the memory position.	S0	S11
S11	The FSM increments the memory addresses respective to the previous node and distance-to-target memory components. If the last address has been reached the FSM advances to S1 and finalizes signal initializations by writing the distance value of the source intersection to zero.	S10	S10; S1
S12	In this state the FSM increments the distance component's address and advances to S2. If the last address has been reached it advances to S3.	S2	S2; S3

User Input Controller

Content Addressable Memory (CAM) components have been mentioned earlier in this paper but have not been defined and their operation has not been explained yet. Due to the dependent nature the following controller has on this component it is useful to define it here.

Unlike a normal memory component (like the RAM memories implemented previously), where an address value is supplied to the memory in order to obtain a data word, in a CAM memory the data word is supplied and the correspondent storage address is returned. In this work a CAM memory is used to store the intersections tile coordinates in order to support the traffic lights VGA output as will be discussed in section 3.6. Another utility for this CAM is discussed next.

Since for this section we need to identify on which intersection the user clicks on, a CAM memory with the intersection's tile coordinates presents itself as a feasible solution. An alternative method is searching each intersection comparing its tile position coordinates with the current mouse coordinates, but since (as will be seen further on) we already have a CAM memory with the intersection position coordinates it would be more efficient to somehow share this memory component limiting extra hardware components. We mention tile coordinates here since each tile is 16-by-16 pixels and fits exactly to each intersection square. Therefore we can discard the four least significant bits of the mouse coordinates, for both the x and y coordinates. The CAM has for data input the concatenation between the current vertical and horizontal pixel coordinates (discarding the four less significant bits of each) driven from the VGA synchronization circuit, since these pixel coordinates sweep the entire screen in order to assess if the user clicks on a valid intersection an indirect comparison can be made. This comparison consists in comparing the mouse coordinates with the pixel coordinates, when these values match the output of the CAM memory is read in order to identify a possible intersection identification.

As stated before the intersection on which the user click is considered as the source intersection and the vehicle's current destination intersection is considered as the target intersection according to the presented Dijkstra's algorithm pseudo-code. The user input control therefore sets these signals and gives clearance to the Dijkstra calculation FSM to start.

VHDL Description

The following state chart symbolizes the User Input Controller's FSM operation (Figure 3.5.3-8):

State	State Description	Previous State	Next State
S0	The FSM waits until there is a left mouse click	Reset; S1; S6	S1
S1	The FSM waits until the mouse coordinates equal the pixel coordinates, once this happens the intersection position CAM is read to evaluate a possible match, and in case there is a match the FSM transitions to S2 otherwise it reverts to S1	S0	S0; S2
S2	The FSM writes the CAM intersection match address to the source signal	S1	S3
S3	The Dijkstra algorithm idle signal is read, if this signal is high this means the Dijkstra FSM is ready and therefore the user input controller writes the target signal as the current user controlled vehicle's destination intersection and brings the dij_go signal high giving permission for the Dijkstra FSM to advance. If the Dijkstra FSM is idle the FSM advances to S4 otherwise it remains in S3 until the idle signal is high.	S2; S3	S4
S4	The FSM waits until the Dijkstra's algorithm completes and advances to state S5.	S3; S4	S5
S5	Since the Dijkstra's algorithm has completed the SPF calculation the user input controller gives permission for the user controlled vehicle to advance by bringing the uveh_go signal high, the FSM then advances to S6	S4	S6
S6	The FSM waits until the user controlled vehicle follows the shortest path and arrives at the target intersection.	S5; S6	S0

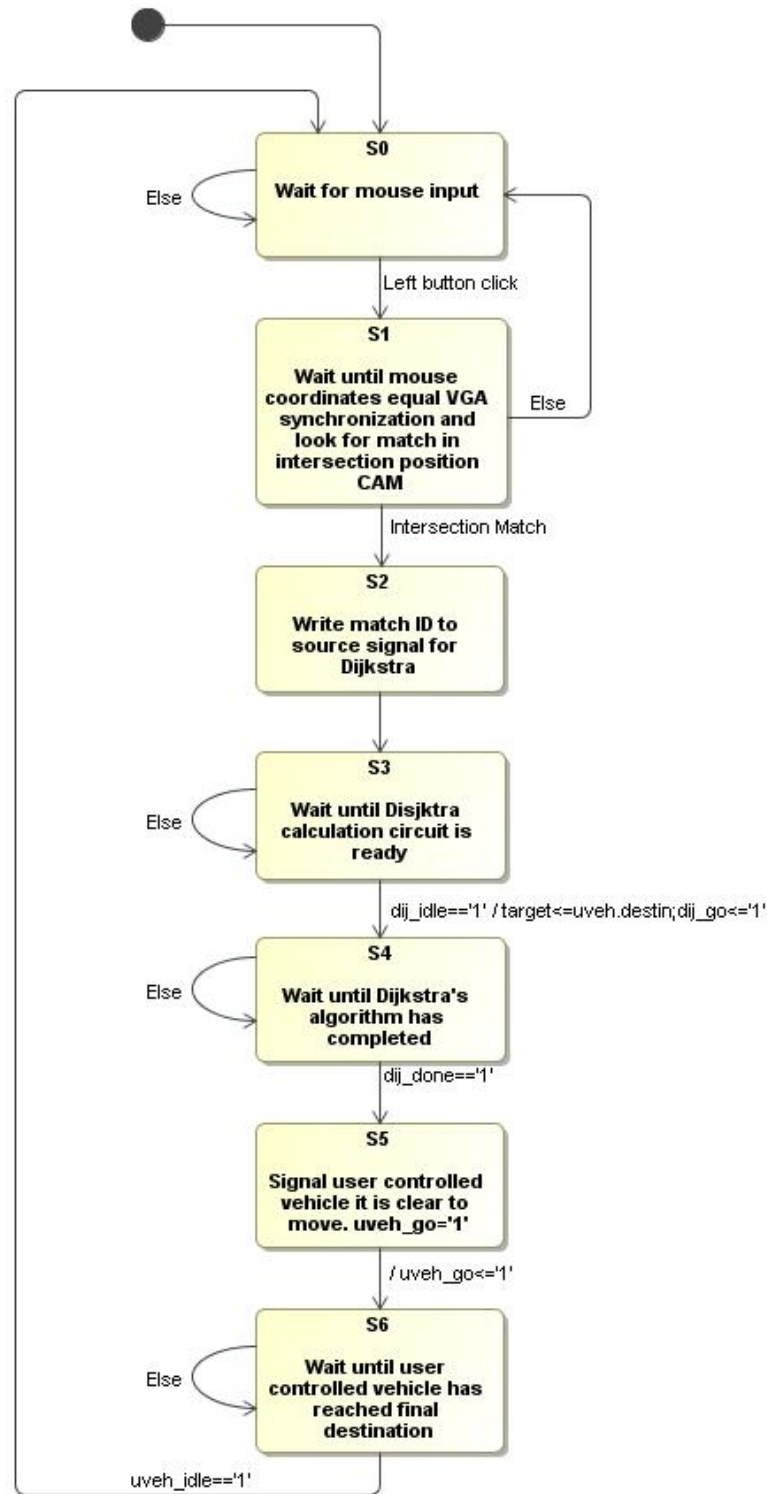


Figure 3.5.3-8 User Input Controller state chart

3.6 USER INTERFACE

One of the current work's interests is giving the user the ability to change UTS simulation related parameters in real-time, this offers many advantages since having to re-synthesize a design and transferring it to the FPGA board every time a bug is found can take a considerable amount of time, especially when network complexity arises and the synthesis and implementation process can take up to several hours. As discussed earlier this interaction is built around the PS/2 mouse and graphical buttons which are drawn in a bitmap. Therefore this bitmap has two functions, one is the graphical representation of the UTS network and to draw user interface buttons with which the user can interact by clicking on and the other function is to serve as a basic debugging interface allowing the user to monitor several signals.

Regarding the user changeable parameters, this paper focuses in the following parameters:

- Destination intersection of each route.
- Status (On/Off).
- Current timer value.
- “Intelligent” sensing status (On/Off).
- Timeout values for the Red, Yellow and Green status.
- Current traffic light status (Red, Yellow or Green)

Consequently the user has to provide information of the desired route and corresponding intersection in order to address a specific route or traffic light.

In regard to the debugging, the following features were considered:

- Auto and Manual clock selection for the User Input, Dijkstra's Algorithm and User Controlled Vehicle finite-state-machines.
- Display the current states for the User Input, Dijkstra's Algorithm and User Controlled Vehicle finite-state-machines.
- Display *source* and *target* intersections signals.
- Access and display the SPF related previous node and "distances" memory components.

Using Adobe Photoshop as the graphics editing program the following user interface was designed, where the first figure (Figure 3.6-1) represents the default screen with user input buttons to the right. Figure 3.6-2 illustrates the user interface when the user click on the "INTERSECTIONS" labeled button to the right. Finally figure 3.6-3 illustrates the debugging interface where the user can monitor the user controller vehicle related signals. The user can then return to the default screen by clicking on the "CLS" button. The simulation continues executing in all screens. The offset mechanism discussed in section 3.2 for the flash memory is responsible for loading a different bitmap according to the user input.

In the intersections interface each intersection has been numbered accordingly allowing an easy access for the user. In the debugger interface besides being numbered, the separation between each tile is evidenced in order to allow the route lengths to be counted easily.



Figure 3.6-1 Default screen with user interface menu to the right



Figure 3.6-2 User "INTERSECTIONS" interface

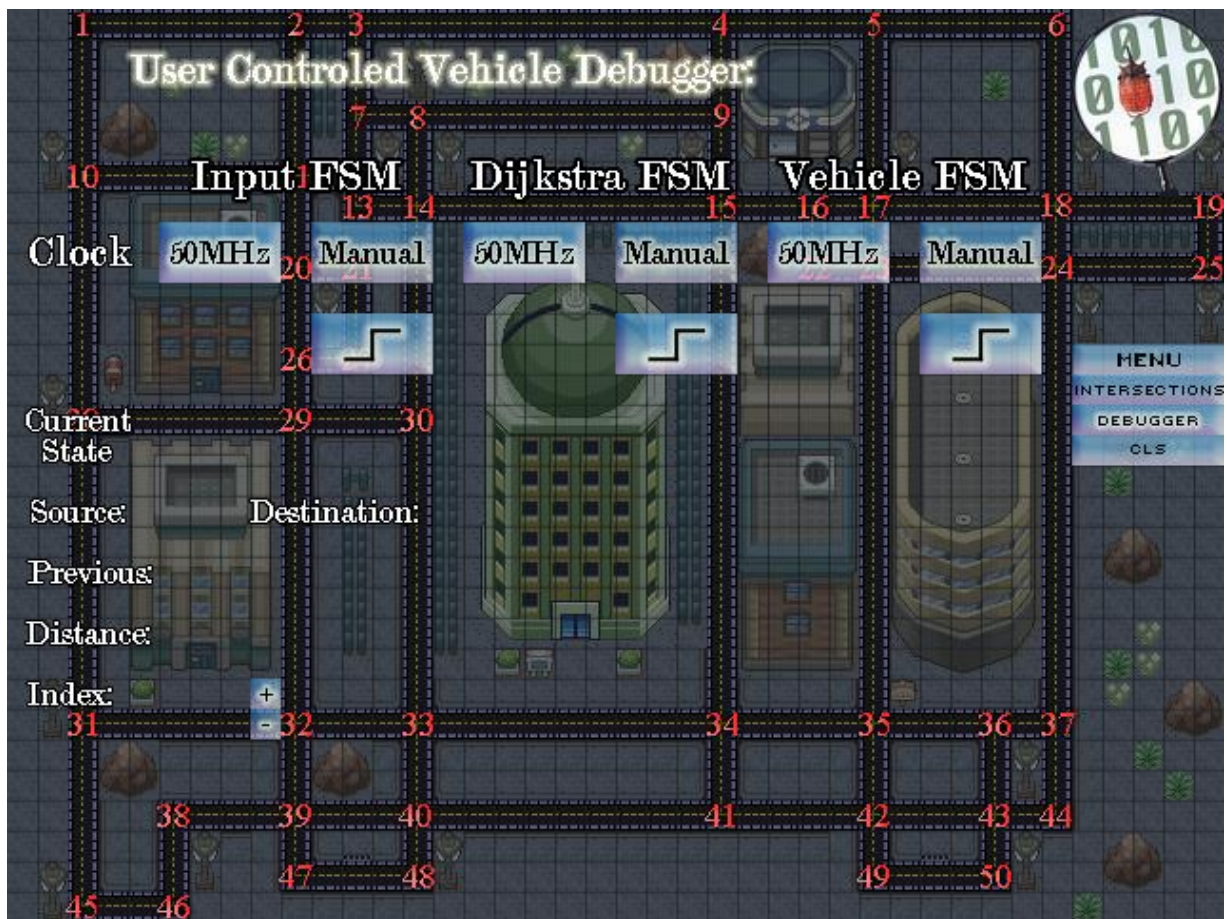


Figure 3.6-3 User “DEBUGGER” interface

The intersection number and intersection route fields are straight forward, allowing the correct route to be addressed. By providing the intersection numbering in the background (figure 3.6-2) the user can then specify the intersection as intended. Once both intersection and route have been specified the user interface controller has enough information to address the respective route and traffic light memory components and signals. This initially presented a difficulty since we cannot overwrite these signals as this would infer that most of these signals (user changeable parameters) had multiple drivers which would obviously damage the FPGA hardware and therefore synthesis naturally fails. A possible solution is integrating the user input functionality in the traffic light controller, developed in section 3.4. In order to obtain this a “load” signal was used, which basically multiplexes two finite state machines inside the same VHDL process (Figure 3.6-4). Additionally a

load signal for each user changeable parameter was also implemented since the user may not wish to edit every parameter. When the *load* signal is set ('1') the finite state machine interrupts its normal functionality (processing traffic lights) and loads the user specified parameters. When the load (*writemode*) signal returns to '0' the traffic light controller resumes processing traffic lights.

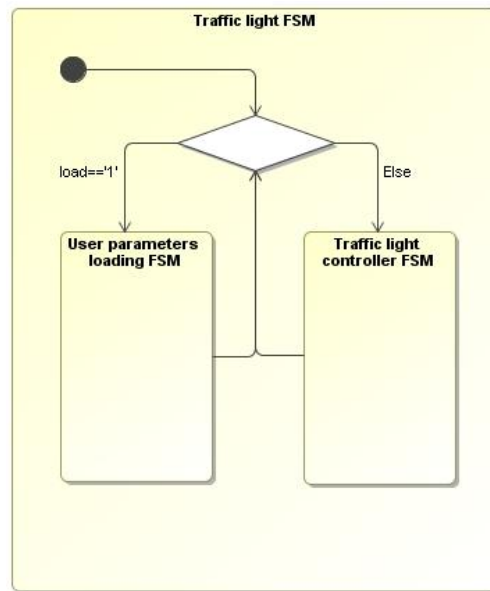


Figure 3.6-4 Traffic Light Controller with user input

Since the user parameters loading FSM is linear we consider its VHDL description (VHDL Description 3.6-1).

```

1 //Inside process
2 if ui_writemode='1' then
3   case TL_loading_state.reg is
4     when s0 =>
5       TL_loading_state.nxt <= s1;
6       If ui_light_stat_apply='1' then
7         net_lights(to_integer(unsigned(ui_inter))).light_on(ui_route_rom(to_integer(unsigned(ui_route)))
8         ) <= ui_light_stat;
9       end if;
10    when s1 =>
11      TL_loading_state.nxt <= s2;

```

```

10     if ui_intelli_apply='1' then
11         intelligent(to_integer(unsigned(ui_inter))) <= ui_intelli;
12     end if;
13     when s2 =>
14         TL_loading_state.nxt <= s3;
15         if ui_timer_apply='1' then
16             watch_addr <= ui_inter & ui_route;
17             watch_din <= ui_timer;
18             watch_we <= "1";
19         end if;
20     when s3 =>
21         TL_loading_state.nxt <= s4;
22         if ui_timers_apply='1' then
23             timeouts_waddr <= ui_inter & ui_route;
24             timeouts_din <= ui_redtimer&ui_yellowtimer&ui_greentimer;
25             timeouts_we <= "1";
26         end if;
27     when s4 =>
28         TL_loading_state.nxt <= s5;
29         if ui_color_apply='1' then
30             if ui_color="00" then
31
32                 netLights(to_integer(unsigned(ui_inter))).status(ui_route_rom(to_integer(unsigned(ui_route))))
33                 <= R;
34             elsif ui_color="01" then
35
36                 netLights(to_integer(unsigned(ui_inter))).status(ui_route_rom(to_integer(unsigned(ui_route))))
37                 <= Y;
38             else
39
40                 netLights(to_integer(unsigned(ui_inter))).status(ui_route_rom(to_integer(unsigned(ui_route))))
41                 <= G;
42             end if;
43         end if;
44     when s5 =>
45         TL_loading_state.nxt <= s0;
46         if ui_dest_interse_apply='1' then
47
48             net_struct(to_integer(unsigned(ui_inter))).neighbor_inter(ui_route_rom(to_integer(unsigned(ui_r

```

```

    oute)))) <= to_integer(unsigned(ui_dest_interse));
42     end if;
43     when others =>
44         NULL;
45     end case;
46 else
47     //Traffic Light Controller
48     ...

```

VHDL Description 3.6-1 User parameters loading FSM

State	State Description	Previous State	Next State
S0	If the load light status user signal is high the FSM writes the user specified data to this value	Reset; S5	S1
S1	If the load intelligent traffic light user signal is high the FSM writes the user specified data to this value	S0	S2
S2	If the load timer user signal is high the FSM writes the user specified data to this value by addressing the timer ram providing the user value as the data input and asserting the correspondent write enable signal.	S1	S3
S3	If the load timeouts user signal is high the FSM writes the user specified data to this value by addressing the timeout ram component providing the user value as the data input and asserting the correspondent write enable signal.	S3	S4
S4	If the load traffic light color status user signal is high the FSM writes the user specified data to this value by decoding the user specified value through a look-up table.	S3	S5
S5	If the load destination intersection user signal is high the FSM writes the user specified data to this value	S4	S0

The FSM repeats this procedure until the *writemode* signal bit is cleared ('0').

Since all interactions are mouse and graphically based, the user has to provide input via the PS/2 mouse, this input is made indirectly which consists in incrementing/decrementing a specific value

by clicking in the correspondent buttons (see figure 3.6-1). In order to implement this, the user interface controller has to monitor the mouse pointer position coordinates comparing it to several button area coordinates. When the user clicks the left mouse button and the mouse is situated inside a button area the controller takes the designated action. In order to identify the mouse cursor's position coordinates we considered, as in previous cases, dividing the screen into 16-by-16 pixel tiles. By using these tiles as the base coordinates and by keeping positioning buttons properly sized and aligned on this tile matrix a simple comparison between the mouse and graphical user interface buttons coordinate regions is needed to detect button clicks. In the following image (Figure 3.6-4) we can observe in what column and rows do each user interface button falls in. By observing the figure it is then easy to describe a controller to execute the desired operation (VHDL Description 3.6-2). These values are then directed not only to the user input FSM mentioned above but also the text-generator block discussed earlier in section 3.2, where they are printed to the screen to provide user feedback

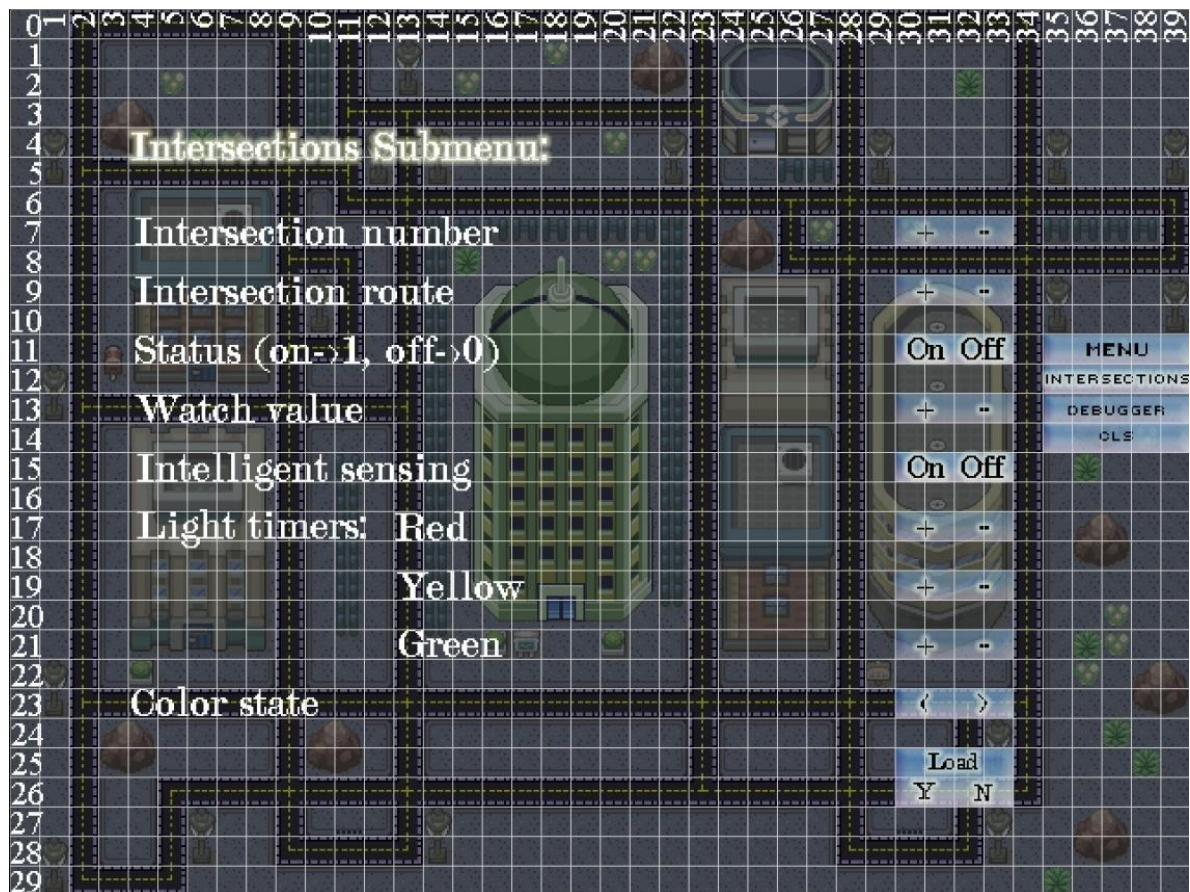


Figure 3.6-5 Screen divided in 16-by-16 pixel tiles in order to localize each button

```

1 -- left button monitor
2 process(mouse_btn)
3 begin
4     if rising_edge(mouse_btn) then
5         -- Menu selection (Flash offset)
6         case mouse_x_int is
7             when 36 to 39 =>
8                 case mouse_y_int is
9                     when 12 =>
10                        offset <= "01";
11                    when 13 =>
12                        offset <= "00";
13                    when others =>
14                        NULL;
15                end case;
16            end case;

```

```

16         when others => NULL;
17     end case;
18     -- Intersections sub menu
19     case offset is
20     when "01" =>
21         case mouse_x_int is
22         when 30 to 31 =>
23             case mouse_y_int is
24             when 7 =>
25                 inter <= inter + 1;
26             when 9 =>
27                 route <= route + 1;
28             when 11 =>
29                 light <= '1';
30             when 13 =>
31                 watch <= watch + 1;
32             when 15 =>
33                 intel <= '1';
34             when 17 =>
35                 rtime <= rtime + 1;
36             when 19 =>
37                 ...

```

VHDL Description 3.6-2 User interface input FSM

3.6.1 TEXT GENERATION CIRCUIT

A general architectural view of the text generation module is presented in figure 3.6.1-1. The character font ROM holds the graphical representation of sixty-four characters, which range from the alphabet, numerical and other symbols. The tile ram divides the screen into 8-by-8 tiles and stores a character symbol for each. The text printing FSM writes information to the tile RAM, this information is provided by other modules as the user input controller or the UTS Network Module were simulation results are read from. Other blocks represent minor logical functions.

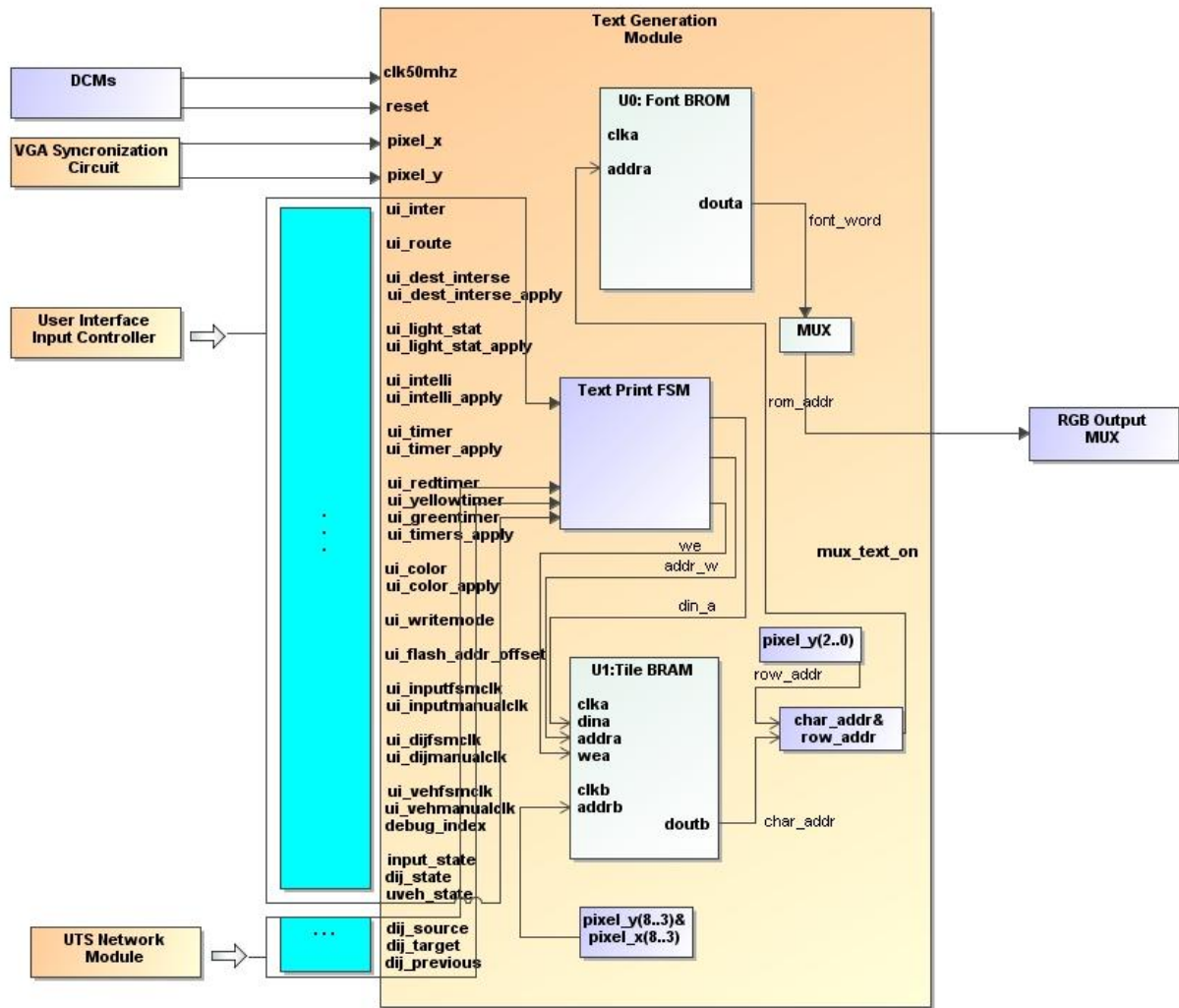


Figure 3.6.1-1 Text Generation Module

The text generation module was developed in order to output the user input and simulation output results, both of which are related to the user interface. The basic text generation circuit scheme is based on the description found in Pong P. Chu. *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version* (291-301) [14] with several modifications in order to lower FPGA resources usage and simultaneously raise the total number of characters that can be displayed on screen. This was accomplished by doing the following:

- Limiting the total number of font characters to 64 in comparison to 128 present in the ASCII code.
- Reducing the font size (or resolution) from 8-by-16 to 8-by-8 pixels.
- Removed cursor that indicated the current text character's position.
- Tile map size set to 64-by-64 characters.
- Using Block Ram for storage instead of distributed memory.

Since the number of font characters is limited to 64, their codes are mostly non ASCII compatible.

Table 3.6.1-1 represents the corresponding custom made character map.

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
@	0	0	P	16	10		32	20	0	48	30
A	1	1	Q	17	11	!	33	21	1	49	31
B	2	2	R	18	12	“	34	22	2	50	32
C	3	3	S	19	13	#	35	23	3	51	33
D	4	4	T	20	14	\$	36	24	4	52	34
E	5	5	U	21	15	:	37	25	5	53	35
F	6	6	V	22	16	&	38	26	6	54	36
G	7	7	W	23	17	‘	39	27	7	55	37
H	8	8	X	24	18	(40	28	8	56	38
I	9	9	Y	25	19)	41	29	9	57	39
J	10	A	Z	26	1A	*	42	2A	A	58	3A
K	11	B	[27	1B	+	43	2B	B	59	3B
L	12	C	→	28	1C	,	44	2C	D	60	3C
M	13	D]	29	1D	-	45	2D	C	61	3D
N	14	E	↑	30	1E	.	46	2E	E	62	3E
O	15	F	←	31	1F	/	47	2F	F	63	3F

Table 3-4 Custom font character map

In order to provide an ASCII code to custom code translation a Matlab function was created that converts a user inputted text string to its corresponding sequence of character codes. The function's source code can be seen in figure 3.6.1-2 and an example conversion output result is demonstrated in figure 3.6.1-3.

```

1  function [out_bin] = conv_text(string)
2
3  for i=1:length(string)
4      if string(i)=='@'
5          tmp = 0;
6      elseif string(i)=='['
7          tmp = 27;
8      elseif string(i)=='>'
9          tmp = 28;
10     elseif string(i)==']'
11         tmp = 29;
12     elseif string(i)=='^'
13         tmp = 30;
14     elseif string(i)=='<'
15         tmp = 31;
16     elseif string(i)==';'
17         tmp = 37;
18     elseif string(i)>='a' && string(i)<='z'
19         tmp = string(i) - 'a' + 1;
20     elseif string(i)>='A' && string(i)<='Z'
21         tmp = string(i) - 'A' + 1;
22     else
23         tmp = string(i);
24     end
25
26     bin = dec2bin(tmp,6);
27     out_bin(i,1)=bin(1);
28     out_bin(i,2)=bin(2);
29     out_bin(i,3)=bin(3);
30     out_bin(i,4)=bin(4);
31     out_bin(i,5)=bin(5);
32     out_bin(i,6)=bin(6);
33 end;

```

Figure 3.6.1-2 ASCII to custom code translation function source code

```

>> out=conv_text('Hello World!')

out =

001000
000101
001100
001100
001111
100000
010111
001111
010010
001100
000100
100001

```

Figure 3.6.1-3 Sample output for ASCII to custom code translation function

Having defined a screen resolution of 640-by-480 pixels this corresponds to an 80-by-60 character tile-map. However by limiting the tile map size to 64-by-64 characters, the tile map read address is defined by a simple concatenation operation between the screen pixel coordinates as illustrated in figure 3.6.1-4, which depicts how the tile RAM is written and read to. Although not overwhelming this simplification avoids further logical functions and lowers resource usage. It must be noted however that since the tile map geometry doesn't follow the screen geometry two major consequences arise. One is that the screen area is only partially occupied in the horizontal direction where there are sixteen missing text columns to complete the screen geometry. In contrast the other consequence is that in the vertical direction there are four overlapping rows of text. The effect of this can be observed in figure 3.6.1-5 where the 512-by-512 pixels square represents the tile-map area and the 640-by-480 pixels square represents the actual screen area. This geometry difference must be had in mind when manipulating this circuit to avoid any misconceptions.

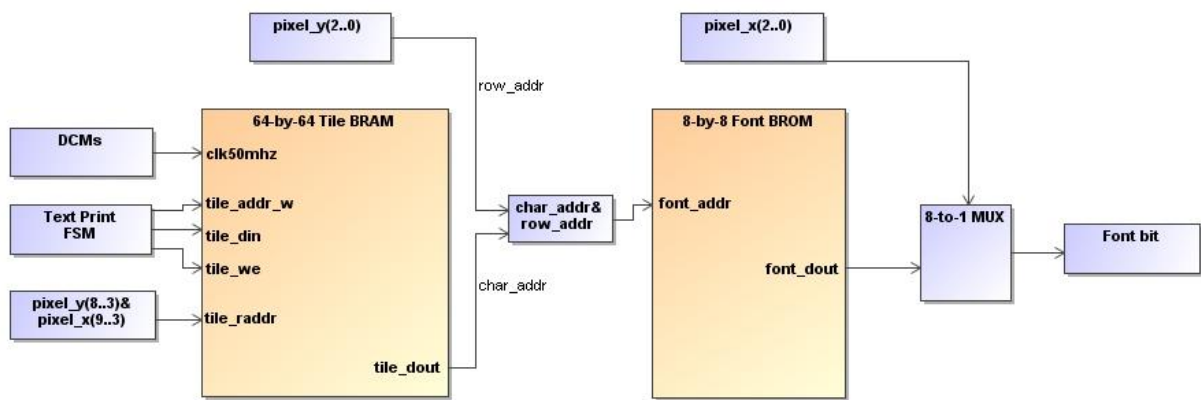


Figure 3.6.1-4 Text generation circuit memory addressing

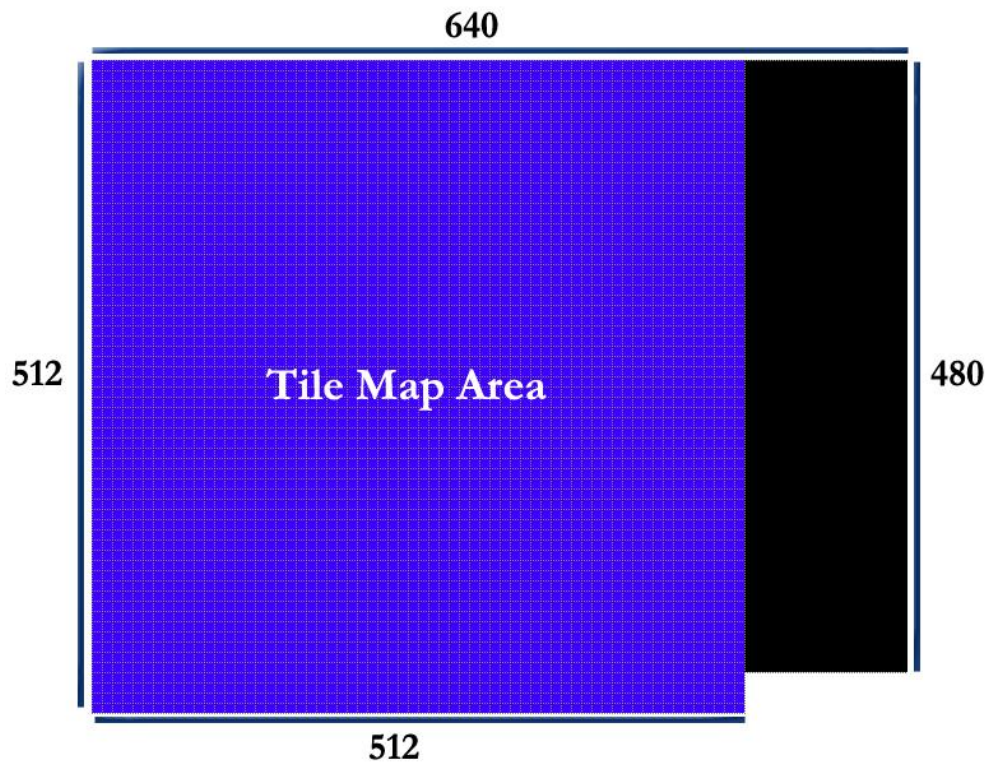


Figure 3.6.1-5 Tile map area overlapping video screen

3.7 VGA OUTPUT COMPONENTS

3.7.1 TRAFFIC LIGHTS

As discussed previously, each intersection holds up to four traffic lights, in order to have a proper VGA output of these we must define an appropriate space for each traffic light inside an intersection. Therefore we consider dividing the intersection into four equal parts (see figure 3.7.1-1) corresponding each to an 8-by-8 pixel tile.

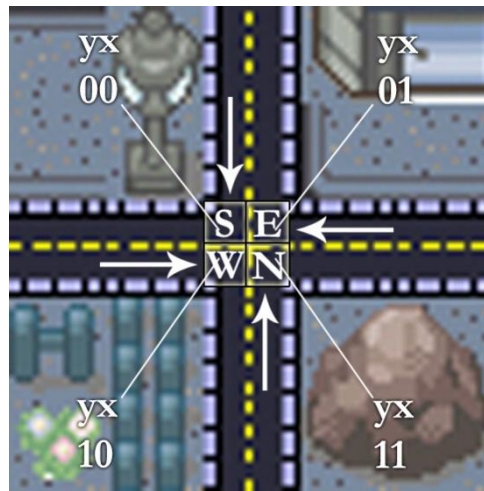


Figure 3.7.1-1 – Intersection traffic light position

In this work the solution to identifying the intersection and outputting the corresponding traffic light color is based on a CAM component. Initially the intention was to implement the traffic light VGA output as a video memory component but due to limited hardware resources (small size of Block RAM and limited performance of the external memory which doesn't allow a fully functioning video ram) a solution was found by utilizing a CAM component (figure 3.7.1-2). This solution is feasible since it only has an operation latency of one system clock cycle which is critical due to the fact that the pixel clock follows tightly behind the system clock and the computation time is limited in order to obtain an acceptable graphical output. Since the entire UTS network was initially drawn based on a 16-by-16 pixel tiles we can implement a CAM that stores these tile

coordinates. By using the horizontal and vertical pixel coordinates from the VGA synchronization circuit as an input to the CAM (the vertical coordinate's five MSBs and horizontal coordinate's six MSBs) and storing the intersection coordinates in the CAM by the same order as they were numbered, each time there is a match the corresponding intersection can be identified by reading the CAM's output. Since the intersection position coordinates are constant we can define the CAM as read-only saving hardware resources. As in other memories, CAM components have a read latency of one clock cycle in this case, but since the pixel clock functions at half the speed of the system clock the graphical displacement should be unnoticeable.

In regard to identifying the individual traffic light areas $pixel_x(3)$ and $pixel_y(3)$ bit signals can be used, which divide the 16-by-16 tiles into four regions as presented in figure 3.7.1-1, where the actual values for these bits is presented. By having identified the region where a certain traffic light should be drawn in the screen it is necessary to verify the traffic light status, in order to assess whether or not it is on/off and what color it currently has in the case its active (on).

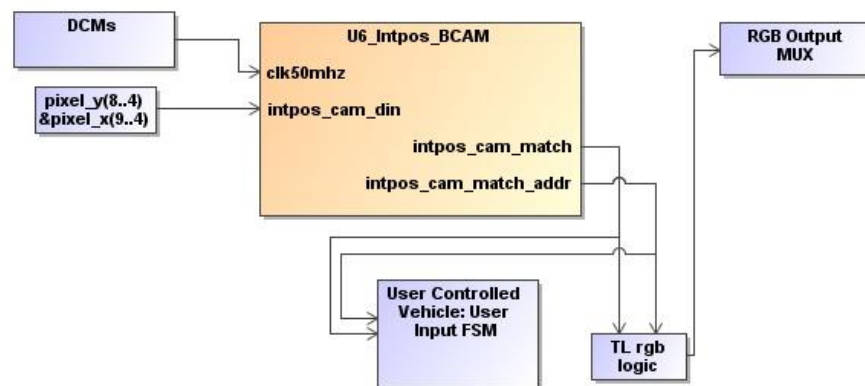


Figure 3.7.1-2 Intersection position coordinates CAM

VHDL Description

The CAM component was implemented using a Xilinx Content Addressable Memory v6.1 IP Core within the IP Coregen utility in Xilinx ISE. More information about this module can be found consulting the datasheet which is accessible through Xilinx ISE. The following description shows

the VHDL signals used to interface this memory component and decode the output to the rgb multiplexer circuit (VHDL Description 3.6.1-1):

```
1 type RGB_LIGHTS_TYPE is array (LIGHTS_ENUM'left to LIGHTS_ENUM'right) of
  std_logic_vector(2 downto 0);
2 constant rgb_lights_rom : RGB_LIGHTS_TYPE := ("100", "110", "010");
3 signal cur_route_plot: ROUTES_ENUM;
4 route_sel <= pix_y(3)&pix_x(3);
5 cur_route_plot <= route_rom(TO_INTEGER(route_sel));
6 intcam_din <= pixel_y(8 downto 4)&pixel_x(9 downto 4);
7 RGB_lights_out <=
  rgb_lights_rom(inter_s_signal(TO_INTEGER(UNSIGNED(intcam_match_addr)+1)).status(
    cur_route_plot))when intcam_match='1' else "000";
```

VHDL Description 3.7.1-1 Intersection position coordinates CAM

The cam component outputs the current intersection when there is a match. The match signal is then used to address the intersections address and the current pixel coordinates are decoded to identify the respective route. In case the current route's traffic light is active the circuit decodes the traffic light color status into a 3-bit word which corresponds to a RGB signal necessary to represent the traffic light's color, this is done via a lookup table (*rgb_lights_rom* in the VHDL description).

3.7.2 VEHICLES

As to simplify each vehicle's graphical representation, 4-by-4 pixel squares are considered to represent vehicles. We previously implemented two separate vehicle models, one ("Dummy" Vehicle) in which normally implements a large number of vehicle's and another model (User controlled vehicle) which implements single vehicle instance. Since the user controlled vehicle is a single instance it doesn't make sense utilizing a CAM component to output this vehicle's representation. Therefore we only resort to a CAM for the dummy vehicle instances. For the user controlled vehicle the position detection can be made in real time with logical functions since a single instance is involved.

VHDL Description

In this case the vehicle related CAM is constantly updated by each vehicle. It isn't important to identify which vehicle is being detected therefore no RGB encoding is needed as all vehicle's can have the same color. The memory block schematic can be seen in figure 3.7.2-1.

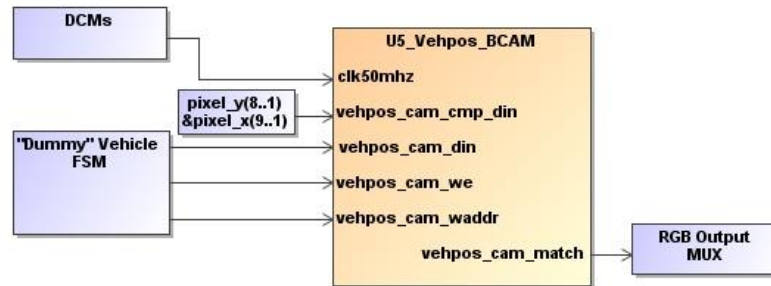


Figure 3.7.2-1 Dummy Vehicle CAM component

As for the user controlled vehicle, VGA output can be accomplished by direct comparison (VHDL Description 3.7.1-1).

```
u_veh_on <= '1' when ((pix_x(9 downto 2)=uveh.position.x(9 downto 2))and(pix_y(8 downto 2)=uveh.position.y(8 downto 2))) else '0';
```

VHDL Description 3.7.2-1 User controlled vehicle VGA output signal

Since we have a number of circuits with a graphical output the following process based on *if-else if* statements was described (VHDL Description 3.7.2-2), which prioritizes VGA the output circuits, as can be observe the mouse cursor has the highest priority and the background bitmap has the lowest.

```

process(clk25m)
begin
  if video_on='0' then
    red <= "000";
    green <= "000";
    blue <= "00";
  elsif mRGB /= "000" then
    red <= mRGB(2) & mRGB(2) & mRGB(2);
    green <= mRGB(1) & mRGB(1) & mRGB(1);
    blue <= mRGB(0) & mRGB(0);
  elsif uveh_on='1' then
    red <= "111";
    green <= "000";
    blue <= "00";
  elsif veh_on='1' then
    red <= "111";
    green <= "111";
    blue <= "00";
  elsif text_on='1' then
    red <= "111";
    green <= "111";
    blue <= "11";
  elsif rgb_lights /= "000" and pixel_x(0)='1' and pixel_y(0)='1' then
    red <= rgb_lights(2) & rgb_lights(2) & rgb_lights(2);
    green <= rgb_lights(1) & rgb_lights(1) & rgb_lights(1);
    blue <= rgb_lights(0) & rgb_lights(0);
  else
    red <= flash_r;
    green <= flash_g;
    blue <= flash_b;
  end if;
end process;

```

VHDL Description 3.7.2-2 RGB output multiplexer

4 DESIGN FLOW

As a final implementation note, it may be useful to acquire a general view of the current work's action and thought. In order to obtain such a status let's consider a situation in which an existing UTS simulation model is available and that this model is composed of the traffic network infrastructure, traffic light and vehicle models. Finally we consider a user, which has a limited knowledge in the systems architecture, that wishes to modify the traffic network's infrastructure and composing model's parameters (such as traffic light timing schemes and vehicle speeds and initial positions).

Since a major objective of this paper is to develop a ready to simulation platform it should be possible for an existing base UTS model description to be edited and executed without an in-depth knowledge of the entire system's architecture, this section describes the work steps necessary in order to implement a UTS based on previously described models.

Considering an existent UTS model description and implementation (an executable design), there are three fundamental simulation constituents that have to be considered:

- Urban Traffic Network's graphical representation and infrastructure Data Set - Through which vehicles navigate and traffic light models situate themselves.
- Traffic Light Data Set - This defines the parameters and specifications necessary for traffic light models to execute.
- Vehicle Data Set - This defines the parameters necessary for existing vehicle models to execute.

In order to edit the traffic network's infrastructure, intersection (traffic light controller) and vehicle model parameters the only alterations that must be made relate directly to these descriptions, or data sets. Figure 4-1 is an activity diagram which represents the suggested design and work flow in order

to edit the traffic network's infrastructure, along with the model parameters of each traffic light controller (aggregated to each intersection) and vehicle behavior of an existing (base) description.

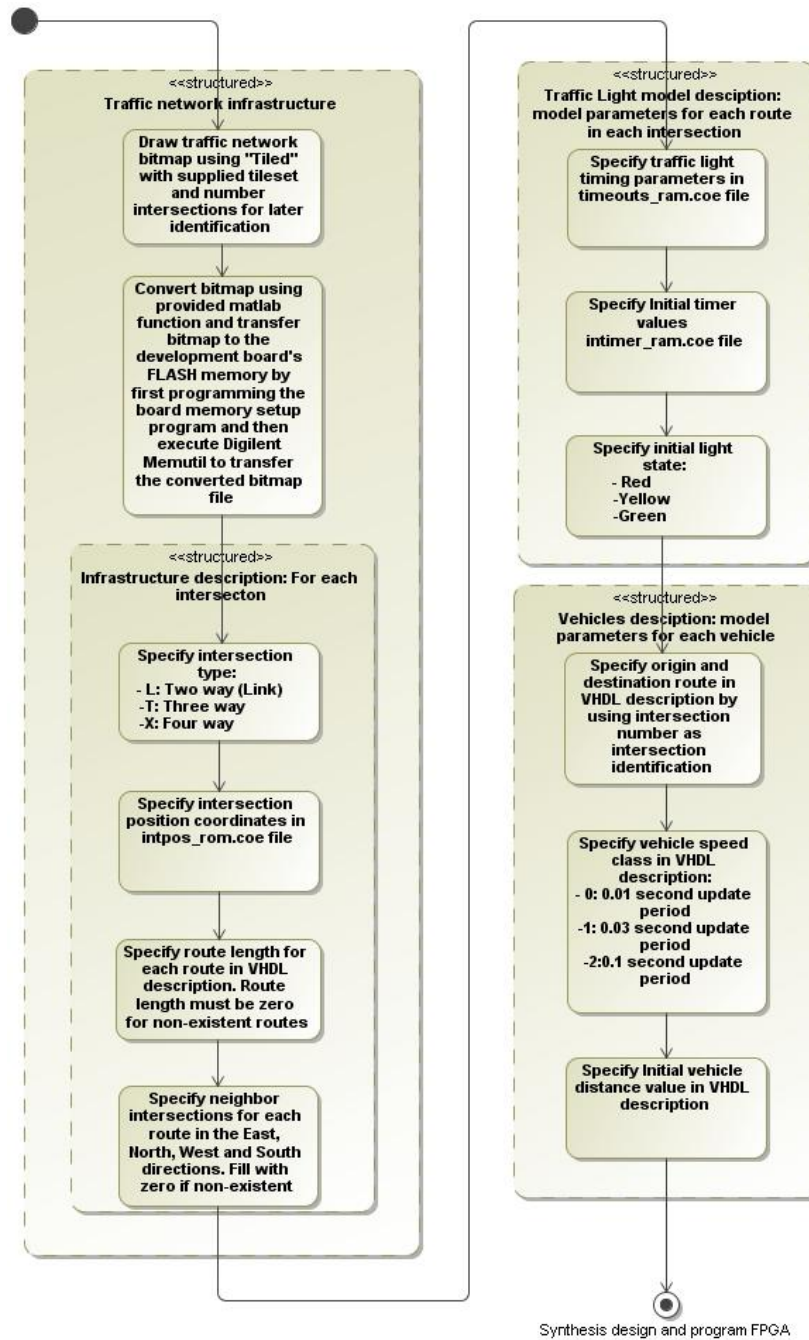


Figure 4-1 Design flow activity diagram

As mentioned above this attempt is only to present a general perspective on the design and work flow necessary to describe an UTS using the base description resultant of this work.

5 IMPLEMENTATION RESULTS & ANALYSIS

5.1 ENCODING

Platform Specifications

Processor: Intel® Core™ 2 Duo E8400

Motherboard: Asus P5KPL AM/PS

System Memory: 4GB DDR2 800Mhz

OS: Windows XP Professional 32-bit SP3

Xilinx ISE 10.1.03

Synthesis and Implementation Options Summary

The defaults as well as the optimal synthesis and implementation settings are identified in table 5.1-1. The optimal settings were mainly achieved by trial-and-error in regard to which settings would allow a larger design to be implemented respecting the 50MHz system clock constraint.

Synthesis Options

Default

Property Name	Value
Optimization Goal	Speed
Optimization Effort	Normal
Use Synthesis Constraints File	<input checked="" type="checkbox"/>
Synthesis Constraints File	
Library Search Order	
Keep Hierarchy	No
Netlist Hierarchy	As Optimized
Global Optimization Goal	AllClockNets
Generate RTL Schematic	Yes
Read Cores	<input checked="" type="checkbox"/>
Cores Search Directories	
Write Timing Constraints	<input type="checkbox"/>
Cross Clock Analysis	<input type="checkbox"/>
Hierarchy Separator	/
Bus Delimiter	<>
Slice Utilization Ratio	100
BRAM Utilization Ratio	100
Case	Maintain
Work Directory	./xst
HDL INI File	
Verilog 2001	<input checked="" type="checkbox"/>
Verilog Include Directories	
Generics, Parameters	
Verilog Macros	
Custom Compile File List	
Other XST Command Line Options	

Optimal

Property Name	Value
Optimization Goal	Speed
Optimization Effort	High
Use Synthesis Constraints File	<input checked="" type="checkbox"/>
Synthesis Constraints File	
Library Search Order	
Keep Hierarchy	No
Netlist Hierarchy	As Optimized
Global Optimization Goal	AllClockNets
Generate RTL Schematic	Yes
Read Cores	<input checked="" type="checkbox"/>
Cores Search Directories	
Write Timing Constraints	<input type="checkbox"/>
Cross Clock Analysis	<input type="checkbox"/>
Hierarchy Separator	/
Bus Delimiter	<>
Slice Utilization Ratio	100
BRAM Utilization Ratio	100
Case	Maintain
Work Directory	./xst
HDL INI File	
Verilog 2001	<input checked="" type="checkbox"/>
Verilog Include Directories	
Generics, Parameters	
Verilog Macros	
Custom Compile File List	
Other XST Command Line Options	

HDL Options

Default

Property Name	Value
FSM Encoding Algorithm	Auto
Safe Implementation	No
Case Implementation Style	None
FSM Style	LUT
RAM Extraction	<input checked="" type="checkbox"/>
RAM Style	Auto
RDM Extraction	<input checked="" type="checkbox"/>
RDM Style	Auto
Automatic BRAM Packing	<input type="checkbox"/>
Mux Extraction	Yes
Mux Style	Auto
Decoder Extraction	<input checked="" type="checkbox"/>
Priority Encoder Extraction	Yes
Shift Register Extraction	<input checked="" type="checkbox"/>
Logical Shifter Extraction	<input checked="" type="checkbox"/>
XDR Collapsing	<input checked="" type="checkbox"/>
Resource Sharing	<input checked="" type="checkbox"/>
Multiplier Style	Auto
Asynchronous To Synchronous	<input type="checkbox"/>

Optimal

Property Name	Value
FSM Encoding Algorithm	Auto
Safe Implementation	No
Case Implementation Style	None
FSM Style	LUT
RAM Extraction	<input checked="" type="checkbox"/>
RAM Style	Auto
RDM Extraction	<input checked="" type="checkbox"/>
RDM Style	Auto
Automatic BRAM Packing	<input type="checkbox"/>
Mux Extraction	Yes
Mux Style	Auto
Decoder Extraction	<input checked="" type="checkbox"/>
Priority Encoder Extraction	Yes
Shift Register Extraction	<input checked="" type="checkbox"/>
Logical Shifter Extraction	<input checked="" type="checkbox"/>
XDR Collapsing	<input checked="" type="checkbox"/>
Resource Sharing	<input checked="" type="checkbox"/>
Multiplier Style	Auto
Asynchronous To Synchronous	<input checked="" type="checkbox"/>

Xilinx Specific Options

Default

Property Name	Value
Add I/O Buffers	<input checked="" type="checkbox"/>
Max Fanout	500
Number of Clock Buffers	24
Register Duplication	<input checked="" type="checkbox"/>
Equivalent Register Removal	<input checked="" type="checkbox"/>
Register Balancing	No
Move First Flip-Flop Stage	<input checked="" type="checkbox"/>
Move Last Flip-Flop Stage	<input checked="" type="checkbox"/>
Pack I/O Registers into IOBs	Auto
Slice Packing	<input checked="" type="checkbox"/>
Use Clock Enable	Yes
Use Synchronous Set	Yes
Use Synchronous Reset	Yes
Optimize Instantiated Primitives	<input type="checkbox"/>

Optimal

Property Name	Value
Add I/O Buffers	<input checked="" type="checkbox"/>
Max Fanout	500
Number of Clock Buffers	24
Register Duplication	<input checked="" type="checkbox"/>
Equivalent Register Removal	<input checked="" type="checkbox"/>
Register Balancing	Yes
Move First Flip-Flop Stage	<input checked="" type="checkbox"/>
Move Last Flip-Flop Stage	<input checked="" type="checkbox"/>
Pack I/O Registers into IOBs	Auto
Slice Packing	<input checked="" type="checkbox"/>
Use Clock Enable	Yes
Use Synchronous Set	Yes
Use Synchronous Reset	Yes
Optimize Instantiated Primitives	<input checked="" type="checkbox"/>

Map Properties

Default

Optimal

Property Name	Value	Property Name	Value
Perform Timing-Driven Packing and Placement	<input type="checkbox"/>	Perform Timing-Driven Packing and Placement	<input checked="" type="checkbox"/>
Map Effort Level	Medium	Map Effort Level	Standard
Extra Effort	None	Extra Effort	Normal
Starting Placer Cost Table (1-100)	1	Starting Placer Cost Table (1-100)	1
Combinatorial Logic Optimization	<input type="checkbox"/>	Combinatorial Logic Optimization	<input checked="" type="checkbox"/>
Register Duplication	<input type="checkbox"/>	Register Duplication	<input type="checkbox"/>
Ignore User Timing Constraints	<input type="checkbox"/>	Ignore User Timing Constraints	<input type="checkbox"/>
Timing Mode	Non Timing Driven	Timing Mode	Non Timing Driven
Trim Unconnected Signals	<input checked="" type="checkbox"/>	Trim Unconnected Signals	<input checked="" type="checkbox"/>
Replicate Logic to Allow Logic Level Reduction	<input checked="" type="checkbox"/>	Replicate Logic to Allow Logic Level Reduction	<input checked="" type="checkbox"/>
Allow Logic Optimization Across Hierarchy	<input type="checkbox"/>	Allow Logic Optimization Across Hierarchy	<input checked="" type="checkbox"/>
Map to Input Functions	4	Map to Input Functions	4
Optimization Strategy (Cover Mode)	Area	Optimization Strategy (Cover Mode)	Area
Generate Detailed MAP Report	<input type="checkbox"/>	Generate Detailed MAP Report	<input type="checkbox"/>
Use RLOC Constraints	<input checked="" type="checkbox"/>	Use RLOC Constraints	<input checked="" type="checkbox"/>
Pack I/O Registers/Latches into IOBs	Off	Pack I/O Registers/Latches into IOBs	Off
Disable Register Ordering	<input type="checkbox"/>	Disable Register Ordering	<input type="checkbox"/>
CLB Pack Factor Percentage	100	CLB Pack Factor Percentage	100
Map Slice Logic into Unused Block RAMs	<input type="checkbox"/>	Map Slice Logic into Unused Block RAMs	<input type="checkbox"/>
Power Reduction	<input type="checkbox"/>	Power Reduction	<input type="checkbox"/>
Power Activity File		Power Activity File	
Other Map Command Line Options		Other Map Command Line Options	

Default

Property Name	Value
Place And Route Mode	Normal Place and Rout
Place & Route Effort Level (Overall)	Standard
Placer Effort Level (Overrides Overall Level)	None
Router Effort Level (Overrides Overall Level)	None
Extra Effort (Highest PAR level only)	None
Starting Placer Cost Table (1-100)	1
Ignore User Timing Constraints	<input type="checkbox"/>
Timing Mode	Performance Evaluation
Use Bonded I/Os	<input type="checkbox"/>
Generate Asynchronous Delay Report	<input type="checkbox"/>
Generate Clock Region Report	<input type="checkbox"/>
Generate Post-Place & Route Static Timing Report	<input checked="" type="checkbox"/>
Generate Post-Place & Route Simulation Model	<input type="checkbox"/>
Number of PAR Iterations (0-100)	3
Number of Results to Save (0-100)	
Save Results in Directory (.dir will be appended)	
Modelist File (Unix Only)	
Power Reduction	<input type="checkbox"/>
Power Activity File	
Other Place & Route Command Line Options	

Optimal

Property Name	Value
Place And Route Mode	Normal Place and Rout
Place & Route Effort Level (Overall)	High
Placer Effort Level (Overrides Overall Level)	None
Router Effort Level (Overrides Overall Level)	None
Extra Effort (Highest PAR level only)	Normal
Starting Placer Cost Table (1-100)	1
Ignore User Timing Constraints	<input type="checkbox"/>
Timing Mode	Performance Evaluation
Use Bonded I/Os	<input type="checkbox"/>
Generate Asynchronous Delay Report	<input type="checkbox"/>
Generate Clock Region Report	<input type="checkbox"/>
Generate Post-Place & Route Static Timing Report	<input checked="" type="checkbox"/>
Generate Post-Place & Route Simulation Model	<input type="checkbox"/>
Number of PAR Iterations (0-100)	3
Number of Results to Save (0-100)	
Save Results in Directory (.dir will be appended)	
Modelist File (Unix Only)	
Power Reduction	<input type="checkbox"/>
Power Activity File	
Other Place & Route Command Line Options	

Table 5.1-1 Synthesis and Implementation Options

Regarding synthesis options it can be stated that activating register balancing impacts the circuit's maximum operating frequency profoundly and is therefore essential in large scale designs (in relation to the FPGA's capacity).

The design's implementation was separated in two configurations relating to the simulation's content. In the first configuration here presented the user controlled vehicle was omitted in order to verify the scale of implementation that could be achieved with only the "dummy" vehicle model. The reason in doing this is because the user controlled vehicle infers a considerable resource usage due to the user input and the Dijkstra's Algorithm implementation. In the second case we include the user controlled vehicle, hereby limiting the number of "dummy" vehicles for the design to fit. In

both implementations fully configurable traffic light controllers are activated (which accounts to a total number of two hundred traffic lights), as well as the traffic network and traffic lights related user changeable parameters and interface menu. The UTS network infrastructure is the same as presented in figure 3.2.2-6, the characteristics of which are presented in table 5.1-2.

<i>Description</i>	<i>Total</i>
“L” Type Intersections	16
“T” Type Intersections	18
“X” Type Intersections	16
Total Number of Intersections (Nodes)	50
Total Number of Roads (Edges)	74
Total Number of Independently Configurable Traffic Lights	200 (Four per intersection)

Table 5.1-2 UTS network characteristics for network in figure 3.2.2-6

1st Configuration (user controlled vehicle not implemented):

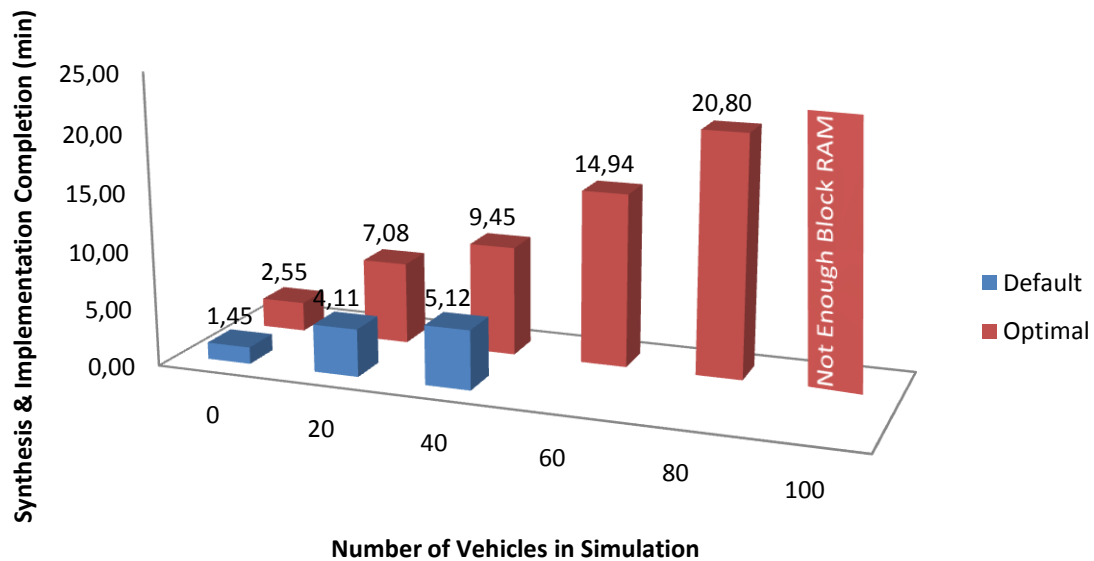


Figure 5.1-1 Synthesis and Implementation completion time in function of the number of vehicles in the UTS simulation

Figure 5.1-1 shows the required completion type (synthesis platform dependent) to implement the design in function of the number of vehicles present in the simulation. Implementation of sixty vehicles upwards fails with default synthesis options, as the number of occupied slices over maps the FPGA's resources. With optimal settings the design is unable to be implemented for a total of one-hundred vehicles due to limited block ram.

It is also useful to access the maximum frequency at which the design can operate properly once again in function of the number of vehicles in the simulation (Figure 5.1-2):

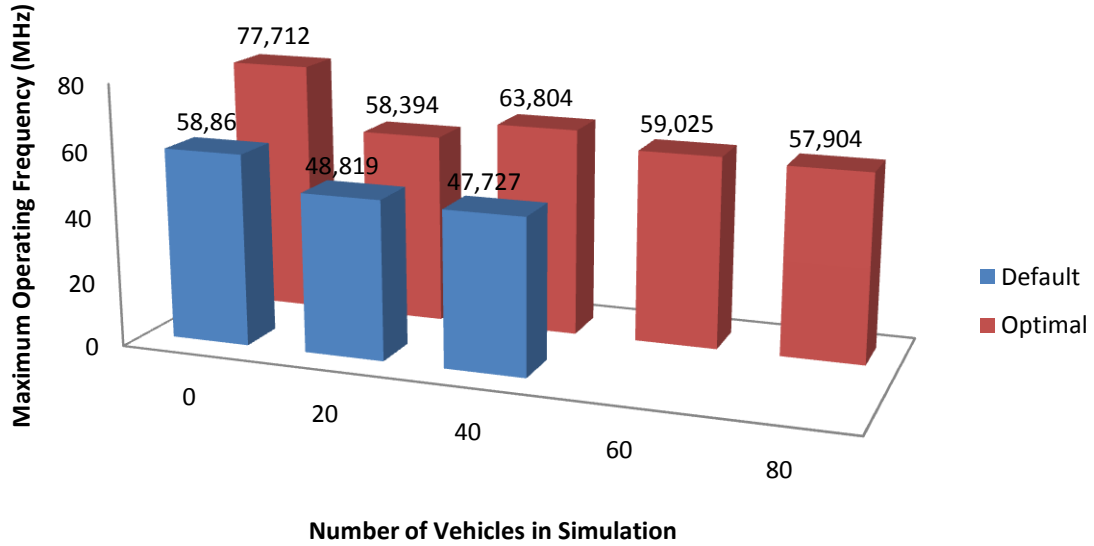


Figure 5.1-2 Maximum design operating frequency in function of the number of vehicles in the UTS simulation

It can be observed that the maximum performance at which the design can operate generally decreases with design size as expected. With default synthesis and implementation options the design shows a below 50MHz clock performance for twenty vehicles upwards, which violates the specified design constraint for the system clock, in practical terms this can lead to an erratic behavior during simulation.

The following results relate the number of vehicles in the simulation with the FPGA's resource usage.

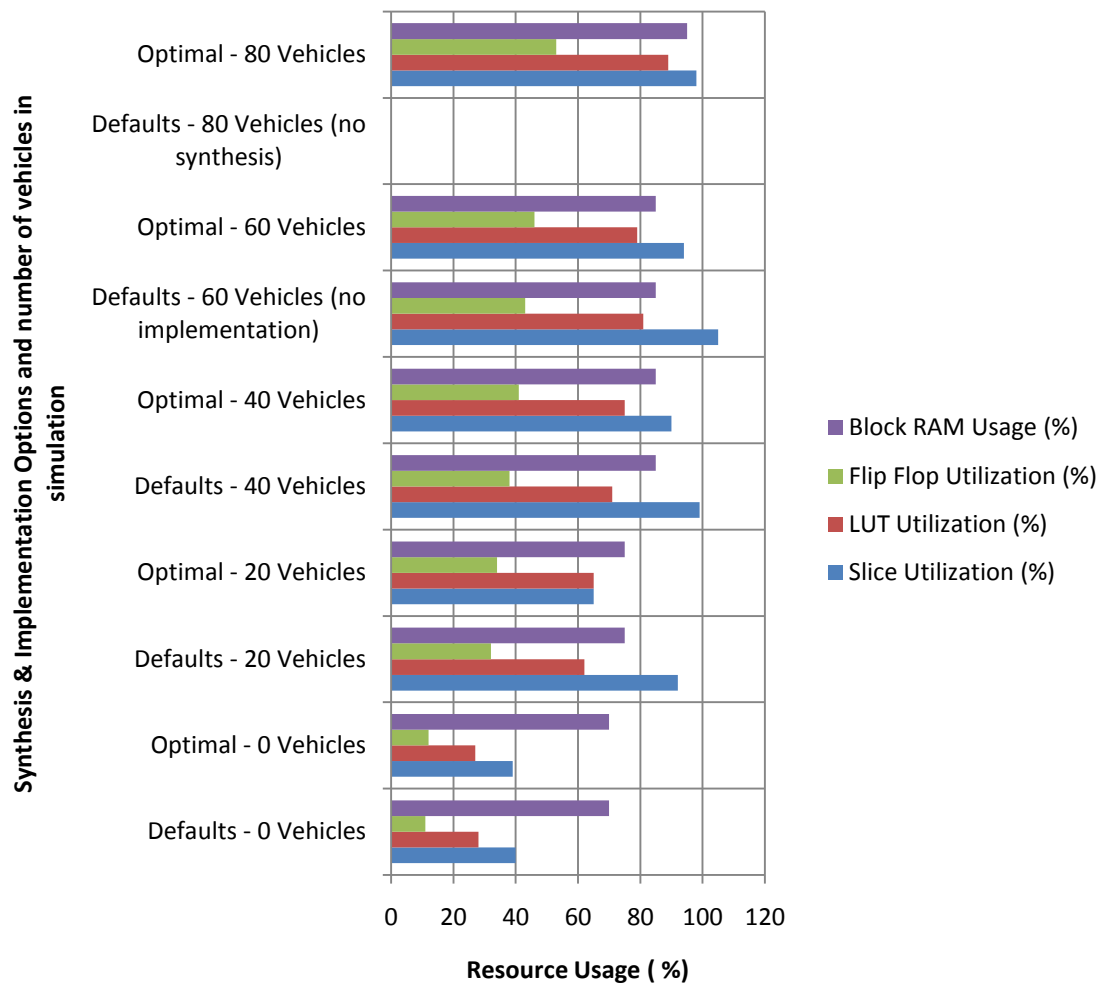


Figure 5.1-3 – Resource usage in function of the number of vehicles in the UTS Simulation

It can be seen that encoding options play an important role when synthesizing and implementing an FPGA design. What limited reaching one-hundred and beyond vehicles was the block ram memory capacity of the FPGA.

5.2 DIJKSTRA'S ALGORITHM

In order to verify the Dijkstra's shortest path finding algorithm implementation a simulation was executed in ModelSim, with a clock signal of 50MHz. Considering the UTS network presented in figure in figure 5.1.1-1 is can be seen that the input refers to the shortest path between intersection two and intersection twenty three.

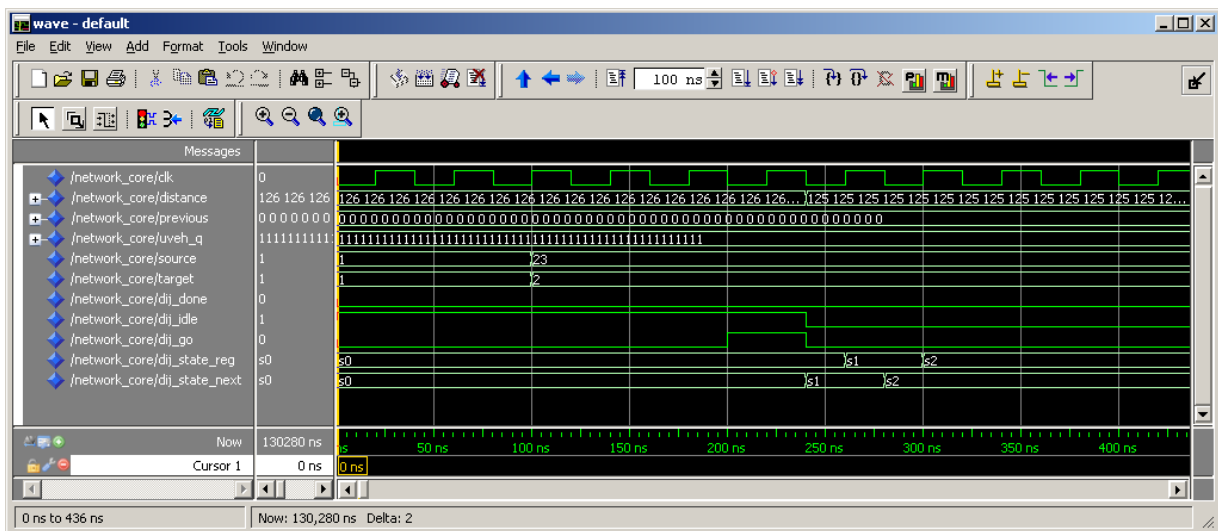


Figure 5.2-1 ModelSim initialization parameters

Figure 5.1.1-2 illustrates the signaling which informs that the computation has completed and in figure 5.1.1-3 the distance result can be seen to be twenty seven tiles, this result can be confirmed by analyzing figure 5.1.1-4, it is important to notice that other paths can be identified with the same distance value, therefore the algorithm provides a single solution. In a practical sense it can also be useful to provide multiple solutions that can be useful as to avoid insisting on traffic jammed arteries.

In terms of performance, the algorithm took 2144 clock cycles to complete which amounts to 85.8 μ S to complete the shortest path computation.

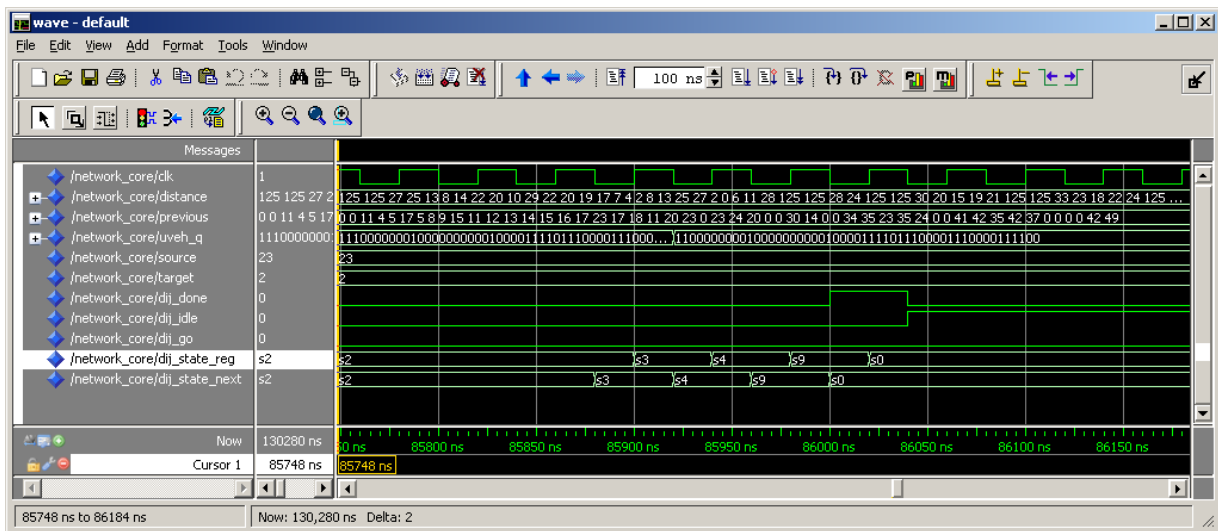


Figure 5.2-2 – Modelsim: Algorithm completion signaled by *dij_done* signal

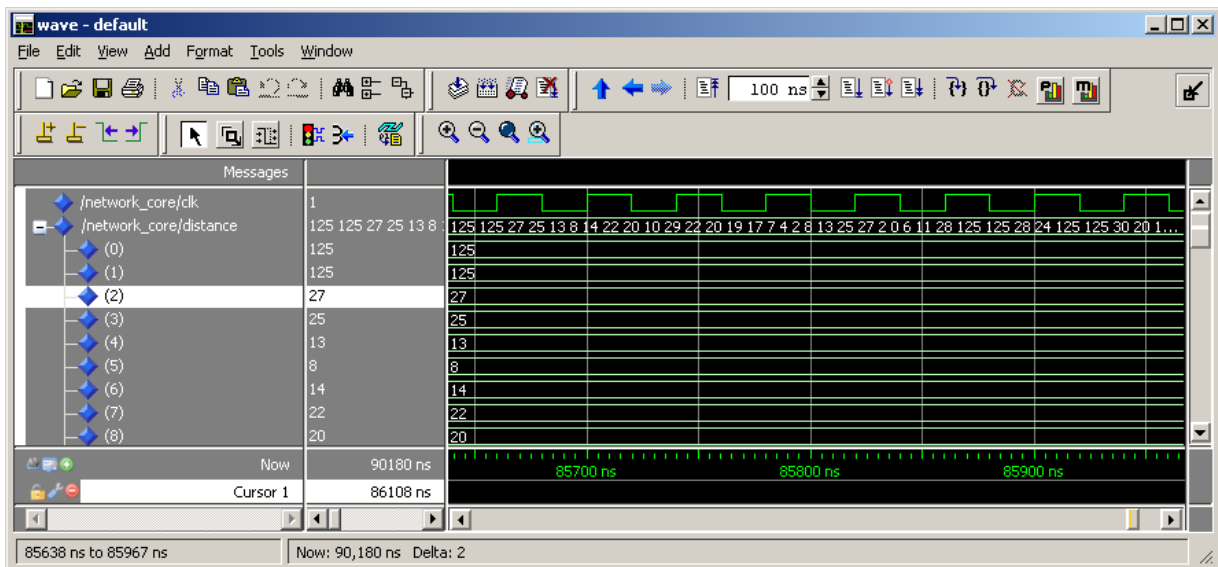


Figure 5.2-3 Modelsim: Distance result of twenty seven tiles

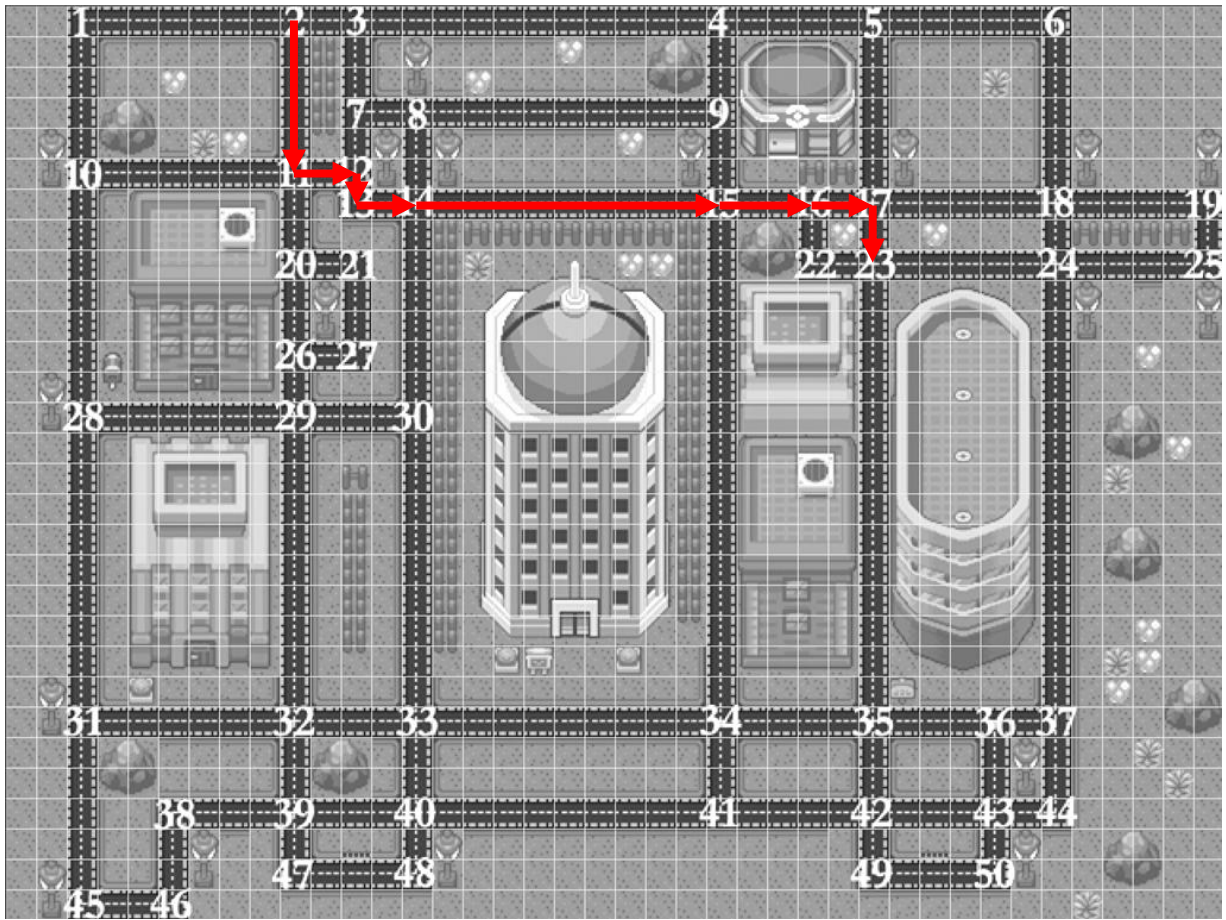


Figure 5.2-4 Illustration of shortest path solution given by Dijkstra's Algorithm implementation

5.3 SIMULATION RESULTS

Although this study does not focus directly on traffic analysis as further development is required, the results should at least resemble those of a real case simulation.

It is therefore useful to analyze the impact the intelligent traffic light model has when compared to the non-sensing traffic light model. By utilizing the first configuration with sixty vehicles no noticeably improvement was observed. This can be due to several factors ranging from poor traffic light coordination or incorrect traffic light timings to unrealistic traffic densities. This fact comes to the same conclusion regarding the previously mentioned (chapter 1 section 4) study from the

University of Strathelyde, which is that the design requires calibration in order to approximate results to a real case scenario.

A satisfying result was observed in regard to vehicle movement and behavior. The network's parameters were also observed to be successfully configurable in real time, allowing a wide range of manipulation of the traffic network's disposition by eliminating routes and traffic light behaviors by being able to edit the corresponding parameter values.

Simulation outputs are illustrated in figures 5.1.2-1, figure 5.1.2-2 and figure 5.1.2-3.



Figure 5.3-1 Design execution: Photograph of simulation with non-sensing traffic lights



Figure 5.3-2 Design execution: Photograph of simulation where intelligent traffic lights have been activated

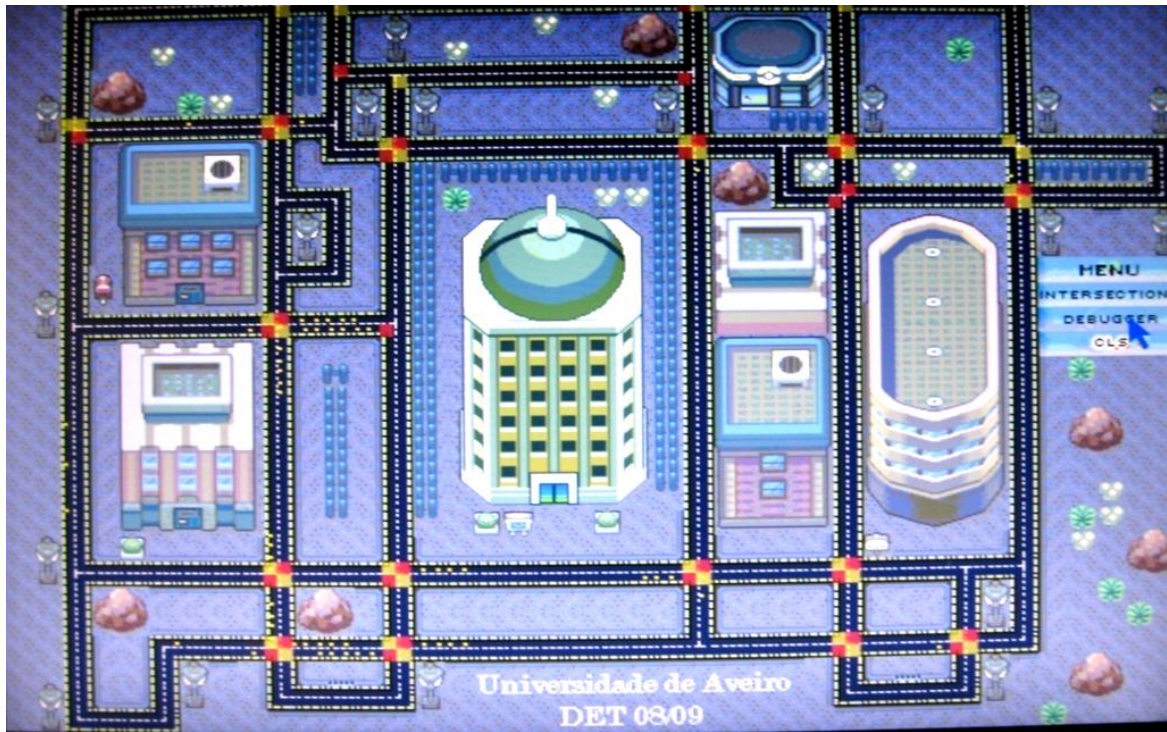


Figure 5.3-3 Design execution: Photograph of simulation, entire screen view with menu and mouse cursor to the right

5.4 DEBUGGING RESULTS

The user controlled vehicle is able to respond correctly to the users input. The finite-state-machine debugging is functional albeit difficult to control since we are debugging three concurrent finite-state-machines with interdependence.

6 CONCLUSION & FUTURE WORK

Conclusions

This paper documents the design and FPGA implementation of an UTS simulation platform which is capable of evaluating urban traffic control elements in a pseudo-real environment when compared to UTS software based implementation. Using FPGA technology provides a unique middle way between software and hardware. As a consequence this work explores UTS Simulation and most importantly Traffic Control Units Evaluation in a novel form, leaving a base implementation and laying out a promising concept.

It has been shown that an UTS Simulation Platform implementation on FPGA hardware is feasible by designing and executing a successful, albeit rudimentary (when compared to advanced software based designs) UTS network composed of traffic light controllers and two vehicle models. User interaction was also achieved through which a user can edit simulation parameters and debug in real time. In limit the current architecture was capable of implementing a total of fifty intersections equipped with fully configurable traffic light controllers which amounts to a total number of two hundred traffic lights. Excluding the user controlled vehicle model a total limit of ninety-six vehicle models were able to be executed. Including the user controlled vehicle this number reduced to twenty-five vehicles plus the user controlled vehicle. Along with real-time user interaction a debugging interface was also implemented.

Although no solid information was extracted from UTS simulation results these were due to lack of network calibration, which falls out of ambit of the current work. Nonetheless a fully functional and interacting result was obtained. Therefore future development and investment should be considered.

Future Work

There are many directions in which the current work could improve upon as a result of further development. In this section we give an overview for future work and ideas and then advance to possible applications of such developments.

This project oversimplifies urban traffic networks as we know them. Since we are aiming at a simulation which pretends to mimic a real world event, is it therefore essential to be able to model a wide range of network configurations and situations. For this we propose the following future developments:

- Extension of the UTS network infrastructure in order to support multi-lane traffic and add support for other traffic junctions (e.g. roundabouts, interchanges). The UTS network infrastructure should also be improved in order to allow multi-direction routes and allow curvilinear routes
- Implement vehicle dynamics, this is essential to simulate vehicle movements correctly as this is a determinant factor responsible for CO2 emissions in traffic jams. Predetermined paths should also be included in the vehicle's model since random route taking is not realistic.
- Implement traffic signals and speed limits.
- Implement statistical data extraction in order to perform UTS analysis. This is an essential aspect as it would allow a quantitative study in traffic phenomena.
- Shortest Path Calculation with configurable parameters such as traffic light of congestion avoidance, in this context this calculation could be considered as a Fastest Path Calculation.

A major limitation of the current work's architecture is related to the graphical output overhead, not only does the current solution occupy a large number of FPGA resources (up to 50% of block ram usage) which would otherwise be useful for logical functions but it has also shown to be cumbersome and scales badly (linearly) with the number of graphical output elements. A more appropriate solution is either to focus on developing a graphics engine (or utilizing commercially available IP-Cores such as the D/AVE 2D engine, from TES Electronic Solutions, <http://www.tesbv.com>) that will execute on the FPGA hardware or to leave graphical output operations to a general purpose computer.

As mentioned throughout chapter three, the current FPGA (Spartan-3E-500) has limited resources and as a consequence the design flexibility and performance suffer. Therefore any further development should be made in a more capable FPGA chip. A table illustrating several Xilinx FPGAs is presented in section 2.2.

The user interface should be implemented recurring not to finite-state-machine but to a microcontroller based architecture such as *picoblaze*. Such would imply a software based behavior and the possibility of using a bus in order to communicate with other modules. This would allow design partitioning and would also lower logical resources used as route-through paths. Design partitioning is a feature present in Xilinx ISE which allows the design to be partially maintained during successive synthesis and implementation processes.

In terms of application besides actual use as a UTS Simulation Platform and Traffic Control Units Evaluation Platform there are other potential uses:

- Autonomous vehicle network simulation platform: Driverless vehicles are truly a history-making event. These vehicles are considered to be integrated into the traffic network, having a constant communication with the UTMC system. The present platform also promises to be a valid to develop and most importantly to evaluate such a ground-breaking technology as autonomous vehicle networks.
- Communications testing platform: By interfacing FPGA hardware with wireless modules a step further in control unit evaluation is made where real life scenario implementations can be studied.
- Since FPGA's are major components in digital image processing, another potential application comes into notice. This application's concept is based in obtaining satellite imagery and processing such images with the objective of identifying traffic networks. These traffic networks can then be processed and various simulations can be held based on the resulting roadmaps. This eliminates the need to insert a traffic network configuration manually.

A . APPENDIX

Although there isn't any out of the box support for graph (as in graph theory) calculations in Matlab there is however a comprehensive number of resources or additional toolboxes that support Shortest Path Calculations in graph structures. One such solution is MatlabBGL, which is a package library for Matlab that provides a complete set of algorithms to work with graphs. The supported algorithms for shortest path finding are the Dijkstra's algorithm, the Bellman-Ford algorithm, Johnson's algorithm, and the Floyd-Warshall algorithm. This package is capable of handling large sparse graphs with thousands of nodes.

In the context of the present work we will therefore give an example of using Matlab to compute shortest path finding based on Dijkstra's algorithm.

Considering as a possibility the use of satellite imagery in which a traffic network of roadmap can be identified, it is therefore useful to identify this roadmap as a graph through which it is possible to perform analysis such as shortest path finding. As an example let's consider a satellite image of an urban area (Barcelona, Spain) which is illustrated in figure A-1. From this image it is possible to identify a traffic network, which can then be structured by numbering its composing intersections as shown in figure A-2.

After identifying a network roadmap each intersection can be interpreted as a node, and each route as an edge therefore composing a graph structure. This data can be plotted as to assure a network match (figure A-3). In order to perform a mathematical analysis in Matlab it is first necessary to translate the graphical data to numerical data. This is possible with image processing but in this case we will enter data manually by identifying the approximate pixel coordinates of each intersection as well as specifying any links it may have with other intersections. After this data is entered it is necessary to construct a matrix to contain every connection between nodes, in this case we consider the graph to be bidirectional (two-way traffic). In order to compute the weighted all pairs shortest path problem we can thereby use the `all_shortest_paths` function of the MatlabBGL library providing as an argument the link identifying matrix. A sample of the output produced is illustrated in figure A-4 and a complete source description is presented in Code Description A-1.



Figure A-1 Satellite image extracted from Google Earth



Figure A-2 – Partial traffic network identified with intersections numbered and routes traced

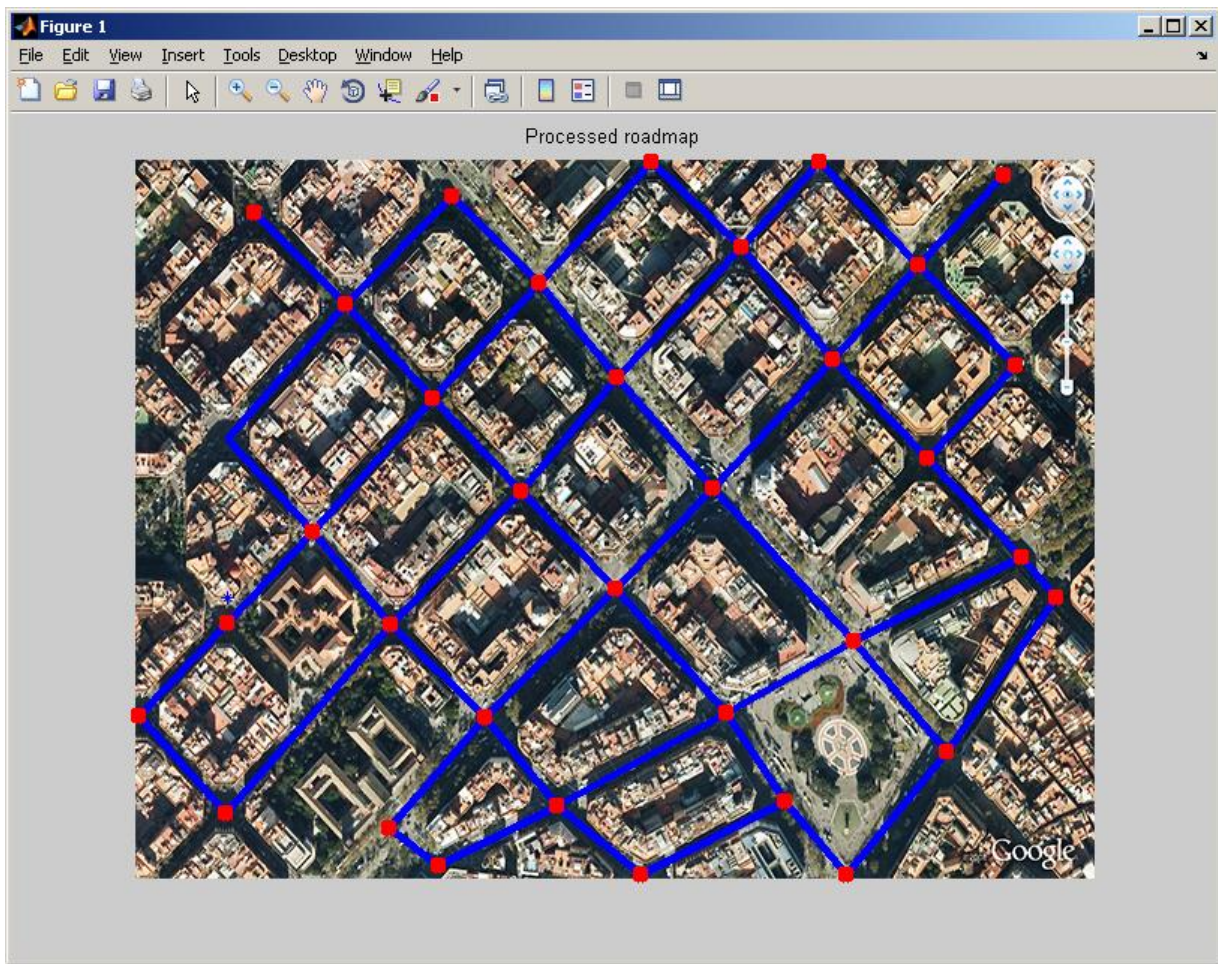


Figure A-3 Matlab output showing data matching with the acquisitioned network imagery

LinkASP =

Columns 1 through 12

0	91	226	288	89	183	314	471	183	285	404	676
91	0	135	379	180	117	248	562	274	231	350	622
226	135	0	514	315	252	252	697	409	366	485	757
288	379	514	0	199	293	424	581	293	395	514	786
89	180	315	199	0	94	225	382	94	196	315	587
183	117	252	293	94	0	131	476	188	114	233	505
314	248	252	424	225	131	0	607	319	245	245	517
471	562	697	581	382	476	607	0	288	390	509	781
183	274	409	293	94	188	319	288	0	102	221	493
285	231	366	395	196	114	245	390	102	0	119	391
404	350	485	514	315	233	245	509	221	119	0	272
676	622	757	786	587	505	517	781	493	391	272	0
504	595	730	614	415	509	640	609	321	423	542	814
282	373	508	392	193	287	418	387	99	201	320	592
383	348	483	493	294	231	362	488	200	117	236	508
501	466	601	611	412	349	480	606	318	235	248	520
760	725	860	870	671	608	739	865	577	494	507	412
1153	1118	1253	1263	1064	1001	1132	1258	970	887	900	793
591	682	817	701	502	596	727	696	408	510	629	901
385	476	611	495	296	390	521	490	202	304	423	695
504	479	614	614	415	362	493	609	321	248	367	639
643	618	753	753	554	501	632	748	460	387	506	778
935	910	1045	1045	846	793	924	1040	752	679	798	1070
1160	1125	1260	1270	1071	1008	1139	1265	977	894	907	812
1552	1517	1652	1662	1463	1400	1531	1657	1369	1286	1299	1204
579	554	689	689	490	437	568	684	396	323	442	714
745	720	855	855	656	603	734	850	562	489	608	880
976	951	1086	1086	887	834	965	1081	793	720	839	1111
494	585	720	604	405	499	630	599	311	413	532	804
154	63	198	442	243	180	311	625	337	294	413	685
587	521	525	697	498	404	273	880	592	518	518	292
1077	1023	1158	1187	988	906	918	1182	894	792	673	401
1665	1630	1765	1775	1576	1513	1644	1770	1482	1399	1412	1305
601	692	827	313	512	606	737	438	606	708	827	1099
641	616	751	751	552	499	630	746	458	385	504	776

Figure A-4 Matlab shortest path finding results sample

```
clear all; close all; clc;
% Reading graphical image
img=imread('Barcelona.jpg');
imshow(img);
hold on

% Shortest Path Finding
%Node (x,y) pixel coordinates
N1=[62 187];
N2=[141 97];
N3=[212 25];
N4=[62 310];
N5=[119 249];
N6=[199 160];
N7=[270 83];
N8=[61 437];
N9=[171 311];
N10=[258 222];
N11=[322 146];
```

```
N12=[405 59];
N13=[170 447];
N14=[234 373];
N15=[321 287];
N16=[386 220];
N17=[466 134];
N18=[523 71];
N19=[203 472];
N20=[282 432];
N21=[395 370];
N22=[480 322];
N23=[592 266];
N24=[529 200];
N25=[588 138];
N26=[434 429];
N27=[542 396];
N28=[615 293];
N29=[338 478];
N30=[80 36];
N31=[345 2];
N32=[457 2];
N33=[580 11];
N34=[3 372];
N35=[475 478];
N36=[0 0];
N37=[0 0];
N38=[0 0];
N39=[0 0];
N40=[0 0];
N41=[0 0];
N42=[0 0];
N43=[0 0];
N44=[0 0];
N45=[0 0];
N46=[0 0];
N47=[0 0];
N48=[0 0];
N49=[0 0];
N50=[0 0];
N51=[0 0];
N52=[0 0];
N53=[0 0];
N54=[0 0];
N55=[0 0];
```



```

N56=[0 0];
N57=[0 0];
N58=[0 0];
N59=[0 0];
N60=[0 0];
N61=[0 0];
N62=[0 0];
N63=[0 0];
N64=[0 0];
    % Node list and plot
List=[N1;N2;N3;N4;N5;N6;N7;N8;N9;N10;N11;N12;N13;N14;N15;N16;N1
7;N18;N19;N20;N21;N22;N23;N24;N25;N26;N27;N28;N29;N30;N31;N32;N
33;N34;N35;N36;N37;N38;N39;N40;N41;N42;N43;N44;N45;N46;N47;N48;
N49;N50;N51;N52;N53;N54;N55;N56;N57;N58;N59;N60;N61;N62;N63;N64
];
    %Identify linked nodes
L=[1,2 ;
1,5 ;
2,3 ;
2,30 ;
2,6 ;
3,7 ;
4,5 ;
4,34 ;
5,6 ;
5,9 ;
6,7 ;
6,10 ;
7,11 ;
7,31 ;
8,34 ;
8,9 ;
9,10 ;
9,14;
10,11 ;
10,15 ;
11,12 ;
11,16 ;
12,31 ;
12,32 ;
12,17 ;
13,14 ;
13,19 ;
14,15 ;

```

```

14,20 ;
15,16 ;
15,21 ;
16,17 ;
16,22 ;
17,18 ;
17,24 ;
18,32;
18,33;
18,25;
19,20;
20,21;
20,29;
21,22;
21,26;
22,23;
22,27;
23,24;
23,28;
24,25;
26,29;
26,35;
27,28;
27,35]
for i=length(L)+1:64
    L(i,:)= [0 0];
end
% Considering a limit of 64 nodes
NLinks=64;
%Compute distance between adjacent nodes
LinkD=sparse(NLinks,NLinks);
for i=1:NLinks
    if ((L(i,1) ~= 0) & (L(i,2) ~= 0))
        LinkD(L(i,1),L(i,2))=round(sqrt((List(L(i,1),1)-
List(L(i,2),1))^2+((List(L(i,1),2)-List(L(i,2),1))^2)));
        plot([List(L(i,1),1) List(L(i,2),1)],
[(List(L(i,1),2)) (List(L(i,2),2))], 'LineWidth',4);
    end
end
plot(List(1,1),480-List(1,2), '*');
AXIS([0 640 0 480]);
for i=2:64
    if ((List(i,1) ~= 0) & (List(i,2) ~= 0))
        plot(List(i,1),List(i,2), 'r*', 'LineWidth',8);
    end
end

```

```
end;  
end  
LinkD=max(LinkD,LinkD');  
Title('Processed roadmap');  
%Perform all shortest path finding computation  
LinkASP=all_shortest_paths(LinkD);  
max(LinkASP(isfinite(LinkASP(:,2)),2))
```

Matlab Script Code A-1 Matlab script in order to compute shortest path finding

B . B I B L I O G R A P H Y

- [1] ETH Zurich/Swiss Federal Institute of Technology (2007, November 16). Self-organized Traffic Light Control System Could Improve Traffic Flow 95 Percent. *ScienceDaily*, Retrieved March 16, 2009, from <http://www.sciencedaily.com/releases/2007/11/071115101810.htm>.
- [2] MANZIE Chris ⁽¹⁾ ; WATSON Harry ⁽¹⁾ ; HALGAMUGE Saman ⁽¹⁾. Fuel economy improvements for urban driving : Hybrid vs. intelligent vehicles. Elsevier, Kidlington, ROYAUME-UNI (1993) (Revue), 1993.
- [3] M. J. Lighthill & G. B. Whitham. *A theory of traffic flow on long crowded roads*. Proceedings of the Royal Society of London, Piccadilly, London, 10 May 1955, **A229**(1178):317–345.
- [4] Michael Schumacher, Laurent Grangier, and Radu Jurca. *Governing Environments for Agent-Based Traffic Simulations*. University of Applied Sciences Western Switzerland Institute of Business Information Systems, Springer-Verlag Berlin Heidelberg 2007.
- [5] Maes, Pattie. "Artificial Life Meets Entertainment: Life like Autonomous Agents," *Communications of the ACM*, 38, 11, 108-114, 1995.
- [6] Matti Pursula. *Transportation engineering: Journal of Geographical information and Decision Analysis*. Vol. 3, no.1, pp. 1-8, 1999.
- [7] Xilinx. *Revolutionary Architecture for the Next Generation Platform FPGAs*. Xilinx: Embargoed News, December 8, 2003.
- [8] Xilinx. <http://www.xilinx.com/company/gettingstarted/fpgavsasic.htm> Xilinx: FPGA vs. ASIC. 1994-2008 Xilinx, Inc.
- [9] Roger Hosking, Pentek. *New FPGAs Transform Real-Time System Architectures* . <http://www.rtcmagazine.com/home/article.php?id=101078&pg=1>, February 2009.
- [10] Microsimulation Tools on the WWW. <http://www.its.leeds.ac.uk/projects/smertest/links.html>.
- [11] Russel, Gordon Cowie, Alex McInnes, John Bate, Martin Milne, George. *Simulating Vehicular Traffic Flows using the Cirval System*. Technical Report asiscia-1-94, University of Strathclyde, May 1994.
- [12] Russel, Gordon Shaw, Paul McInnes, John Ferguson, Neil . *Rapid Simulation of Urban Traffic using FPGAs*. Technical Report asiscia-2-94 University of Strathclyde, May 1994.
- [13] Justin L. Tripp, Henning S. Mortveit, Anders A. Hansson, Maya Gokhale. *Metropolitan Road Traffic Simulation on FPGAs*. Los Alamos National Library NM87545, 2005.
- [14] Pong P. Chu. *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*. Wiley-Interscience, Feb 4, 2008.
- [15] Shortest Path Finding Algorithms - Wikipedia. http://en.wikipedia.org/wiki/Shortest_path_problem.
- [16] Richard E. Haskell, Darrin M. Hanna. *Learning By Example Using VHDL, Advanced Digital Design With a NEXYS2 FPGA Board*. LBE Books, Rochester, Michigan, 2008.
- [17]. *Skhyaron, Valery*. http://ieeta.pt/~skl/SDR_V. University of Aveiro, department of Electronics, Telecommunications and Informatics.
- [18]. *Block Memory Generator V2.8, DS512*. Xilinx, September 19, 2008.
- [19]. *Using Block RAM in Spartan-3 Generation FPGAs*. Xilinx, XAPP463, March 1 2005.
- [20]. *PicoBlaze 8-bit Embedded Microcontroller User Guide: For Spartan-3, Virtex-II, and Virtex-II Pro FPGAs*. Xilinx UG129, June 24, 2008.
- [21]. <http://www.geocities.com/SiliconValley/Screen/2257/vhdl/lfsr/lfsr.html>.
- [22]. *Dijkstra's Algorithm*, From Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm. Wikipedia, April 2009.
- [23]. *ISE Help: FPGA Design Flow Overview* http://www.xilinx.com/itp/xilinx8/help/iseguide/html/ise_fpga_design_flow_overview.htm. Xilinx, inc. 1995-2005.
- [24]. *VHDL Standard www.vhdl.org*. The Electronic Design Automation (EDA) and Electronic Computer-Aided Design (ECAD), Copyright (c) 1994-2006 by Accellera.